Designing database applications

Topic groups See also

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

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

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

The following topics introduce common considerations when designing a database application:

- Using databases
- Database architecture
- Designing the user interface

Using databases

Topic groups See also

The components on the Data Access page, the ADO page, or the InterBase page of the Component palette allow your application to read from and write to databases. The components on the Data Access page use the Borland Database Engine (BDE) to access database information which they make available to the data-aware controls in your user interface. The components on the ADO page use ActiveX Data Objects (ADO) to access the database information through OLEDB. The components on the InterBase page access an InterBase database directly.

Depending on your version of Delphi, the BDE includes drivers for different types of databases. While all types of databases contain tables which store information, different types support additional features such as

- Database security.
- Transactions.
- Data dictionary.
- Referential integrity, stored procedures, and triggers.

Types of databases

Topic groups See also

You can connect to different types of databases, depending on what drivers you have installed with the Borland Database Engine (BDE) or ActiveX Data Objects (ADO).

These drivers may connect your application to local databases such as Paradox, Access, and dBASE or remote database servers like Microsoft SQL Server, Oracle, and Informix. Similarly, the InterBase Express components can access either a local or remote version of InterBase.

Note: Different versions of Delphi come with the components that use these drivers (BDE or ADO), or with the InterBase express components.

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

- How much data will the tables hold?
- How many users will be sharing these tables?
- What type of performance (speed) do you require from the database?

Local databases

Topic groups See also

Local databases reside on your local drive or on a local area network. They have proprietary APIs for accessing the data. Often, they are dedicated to a single system. When they are shared by several users, they use file-based locking mechanisms. Because of this, they are sometimes called file-based databases.

Local databases can be faster than remote database servers because they often reside on the same system as the database application.

Because they are file-based, local databases are more limited than remote database servers in the amount of data they can store. Therefore, in deciding whether to use a local database, you must consider how much data the tables are expected to hold.

Applications that use local databases are called single-tiered applications because the application and the database share a single file system.

Examples of local databases include Paradox, dBASE, FoxPro, and Access.

Remote database servers

Topic groups See also

Remote database servers usually reside on a remote machine. They use Structured Query Language (SQL) to enable clients to access the data. Because of this, they are sometimes called SQL servers. (Another name is Remote Database Management system, or RDBMS.) In addition to the common commands that make up SQL, most remote database servers support a unique “dialect” of SQL.

Remote database servers are designed for access by several users at the same time. Instead of a file-based locking system such as those employed by local databases, they provide more sophisticated multi-user support, based on transactions.

Remote database servers hold more data than local databases. Sometimes, the data from a remote database server does not even reside on a single machine, but is distributed over several servers.

Applications that use remote database servers are called two-tiered applications or multi-tiered applications because the application and the database operate on independent systems (or tiers).

Examples of SQL servers include InterBase, Oracle, Sybase, Informix, Microsoft SQL server, and DB2.

Database security

Topic groups See also

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

Most SQL servers require a password and user name to use the database server at all. Once the user has logged in to the database, that username and password determine which tables can be used. For information on providing passwords to SQL servers when using the BDE, see Controlling server login. For information on providing this information when using ActiveX Data Objects (ADO), see Controlling the connection login. For information on providing this information when using the InterBase direct access components, see the *OnLogin* event of *TIBDatabase*.

Note that not all editions of Delphi include the direct access ADO and InterBase components.

When designing database applications, you must consider what type of authentication is required by your database server. If you do not want your users to have to provide a password, you must either use a database that does not require one or you must provide the password and username to the server programmatically. When providing the password programmatically, care must be taken that security can't be breached by reading the password from the application.

If you are requiring your user to supply a password, you must consider when the password is required. If you are using a local database but intend to scale up to a larger SQL server later, you may want to prompt for the password before you access the table, even though it is not required until then.

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

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

Transactions

Topic groups See also

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

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

Although transaction support is not part of most local databases, the BDE drivers provide limited transaction support for some of these databases. For SQL servers and ODBC-compliant databases, the database transaction support is provided by the component that represents the database connection. In multi-tiered applications, you can create transactions that include actions other than database operations or that span multiple databases.

For details on using transactions in BDE-based database applications, see Using transactions. For details on using transactions in ADO-based database applications, see Working with (connection) transactions. For details on using transactions in multi-tiered applications, see Managing transactions in multi-tiered applications. For details on using transactions in applications that use the InterBase direct access components, see the *TIBTransaction* component.

The Data Dictionary

Topic groups See also

When you use the BDE to access your data, your application has access to the Data Dictionary. The Data Dictionary provides a customizable storage area, independent of your applications, where you can create extended field attribute sets that describe the content and appearance of data.

For example, if you frequently develop financial applications, you may create a number of specialized field attribute sets describing different display formats for currency. When you create datasets for your application at design time, rather than using the Object Inspector to set the currency fields in each dataset by hand, you can associate those fields with an extended field attribute set in the data dictionary. Using the data dictionary ensures a consistent data appearance within and across the applications you create.

In a client/server environment, the Data Dictionary can reside on a remote server for additional sharing of information.

To learn how to create extended field attribute sets from the Fields editor at design time, and how to associate them with fields throughout the datasets in your application, see Creating attribute sets for field components. To learn more about creating a data dictionary and extended field attributes with the SQL and Database Explorers, see their respective online help files.

A programming interface to the Data Dictionary is available in the drintf unit (located in the lib directory). This interface supplies the following methods:

Routine	Use
DictionaryActive	Indicates if the data dictionary is active.
DictionaryDeactivate	Deactivates the data dictionary.
IsNullID	Indicates whether a given ID is a null ID
FindDatabaseID	Returns the ID for a database given its alias.
FindTableID	Returns the ID for a table in a specified database.
FindFieldID	Returns the ID for a field in a specified table.
FindAttrID	Returns the ID for a named attribute set.
GetAttrName	Returns the name an attribute set given its ID.
GetAttrNames	Executes a callback for each attribute set in the dictionary.
GetAttrID	Returns the ID of the attribute set for a specified field.
NewAttr	Creates a new attribute set from a field component.
UpdateAttr	Updates an attribute set to match the properties of a field.
CreateField	Creates a field component based on stored attributes.
UpdateField	Changes the properties of a field to match a specified attribute set.
AssociateAttr	Associates an attribute set with a given field ID.
UnassociateAttr	Removes an attribute set association for a field ID.
GetControlClass	Returns the control class for a specified attribute ID.
QualifyTableName	Returns a fully qualified table name (qualified by user name).
QualifyTableNameByName	Returns a fully qualified table name (qualified by user name).
HasConstraints	Indicates whether the dataset has constraints in the dictionary.
UpdateConstraints	Updates the imported constraints of a dataset.
UpdateDataset	Updates a dataset to the current settings and constraints in the dictionary.

Referential integrity, stored procedures, and triggers

Topic groups See also

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

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

Database architecture

Topic groups See also

Database applications are built from user interface elements, components that manage the database or databases, and components that represent the data contained by the tables in those databases (datasets). How you organize these pieces is the architecture of your database application.

By isolating database access components in data modules, you can develop forms in your database applications that provide a consistent user interface. By storing links to well-designed forms and data modules in the Object Repository, you and other developers can build on existing foundations rather than starting over from scratch for each new project. Sharing forms and modules also makes it possible for you to develop corporate standards for database access and application interfaces.

Many aspects of the architecture of your database application depend on the type of database you are using, the number of users who will be sharing the database information, and the type of information you are working with.

When writing applications that use information that is not shared among several users, you may want to use a local database in a *single-tiered application*. This approach can have the advantage of speed (because data is stored locally), and does not require the purchase of a separate database server and expensive site licences. However, it is limited in how much data the tables can hold and the number of users your application can support.

Writing a *two-tiered application* provides more multi-user support and lets you use large remote databases that can store far more information.

Note: Support for two-tiered applications requires SQL Links, InterBase, or Microsoft ActiveX Data Objects (ADO).

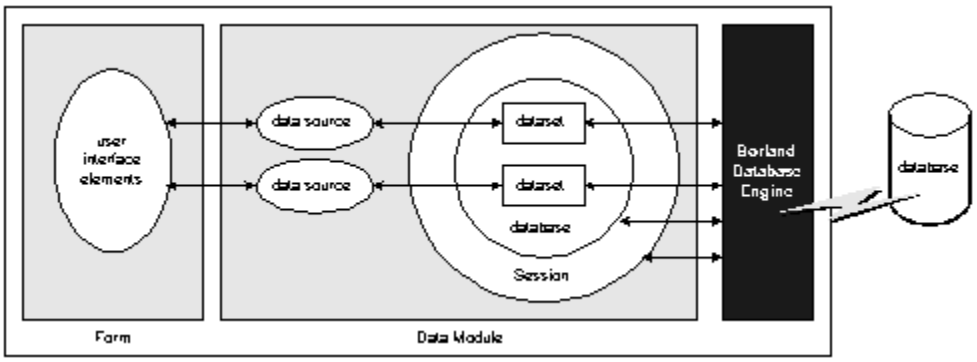
When the database information includes complicated relationships between several tables, or when the number of clients grows, you may want to use a *multi-tiered application*. Multi-tiered applications include middle tiers that centralize the logic which governs your database interactions so that there is centralized control over data relationships. This allows different client applications to use the same data while ensuring that the data logic is consistent. They also allow for smaller client applications because much of the processing is off-loaded onto middle tiers. These smaller client applications are easier to install, configure, and maintain because they do not include the database connectivity software. Multi-tiered applications can also improve performance by spreading the data-processing tasks over several systems.

The development process can get more involved and expensive as the number of tiers increases. Because of this, you may wish to start developing your application as a single-tiered application. As the amount of data, the number of users, and the number of different applications accessing the data grows, you may later need to scale up to a multi-tiered architecture. By planning for scalability, you can protect your development investment when writing a single- or two-tiered application so that the code can be reused as your application grows.

Planning for scalability

Topic groups See also

The VCL data-aware components make it easy to write scalable applications by abstracting the behavior of the database and the data stored by the database. Whether you are writing a single-tiered, two-tiered, or multi-tiered application, you can isolate your user interface from the data access layer as illustrated in the following figure.



A form represents the user interface, and contains data controls and other user interface elements. The data controls in the user interface connect to datasets which represent information from the tables in the database. A data source links the data controls to these datasets. By isolating the data source and datasets in a data module, the form can remain unchanged as you scale your application up. Only the datasets must change.

Note: Some user interface elements require special attention when planning for scalability. For example, different databases enforce security in different ways. See Database security for more information on handling user authentication in a uniform manner as you change databases.

When using Delphi's data access components (whether they use the BDE, ADO, or InterBase Express) it is easy to scale from one-tiered to two-tiered. Only a few properties on the dataset must change to direct the dataset to connect to an SQL server rather than a local database.

A flat-file database application is easily scaled to the client in a multi-tiered application because both architectures use the same client dataset component. In fact, you can write an application that acts as both a flat-file application and a multi-tiered client (see Using the briefcase model).

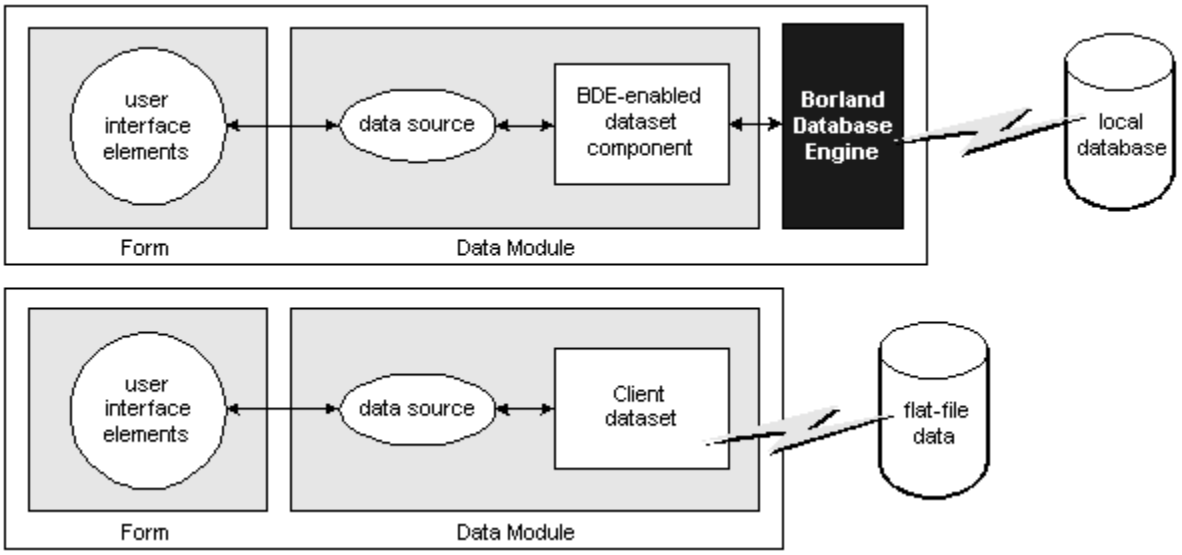
If you plan to scale your application up to a three-tiered architecture eventually, you can write your one- or two-tiered application with that goal in mind. In addition to isolating the user interface, isolate all logic that will eventually reside on the middle tier so that it is easy to replace at a later time. You can even connect your user interface elements to client datasets (used in multi-tiered applications), and connect them to local versions of the InterBase, BDE- or ADO- enabled datasets in a separate data module that will eventually move to the middle tier. If you do not want to introduce this artifice of an extra dataset layer in your one- and two-tiered applications, it is still easy to scale up to a three-tiered application at a later date. See Scaling up to a three-tiered application for more information.

Single-tiered database applications

Topic groups See also

In single-tiered database applications, the application and the database share a single file system. They use local databases or files that store database information in a flat-file format.

A single application comprises the user interface and incorporates the data access mechanism (either the BDE or a system for loading and saving flat-file database information). The type of dataset component used to represent database tables depends on whether the data is stored in a local database (such as Paradox, dBASE, Access, or FoxPro) or in a flat file. The following figure illustrates these two possibilities:

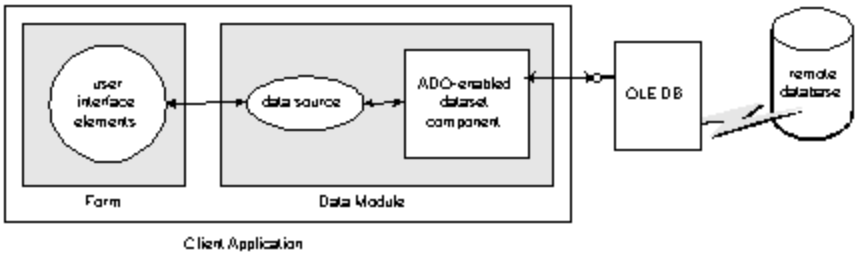


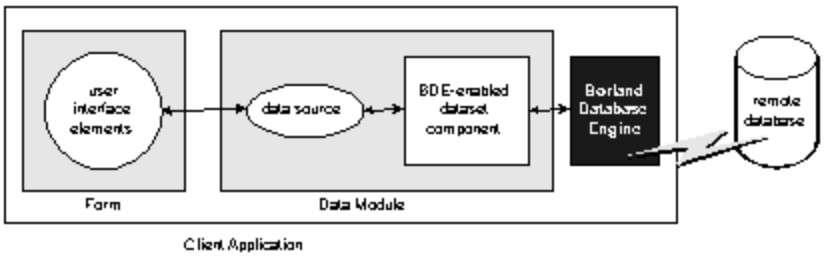
For more information on building single-tiered database applications, see One- and two-tiered applications.

Two-tiered database applications

Topic groups See also

In two-tiered database applications, a client application provides a user interface to data, and interacts directly with a remote database server. The following figure illustrates this relationship.





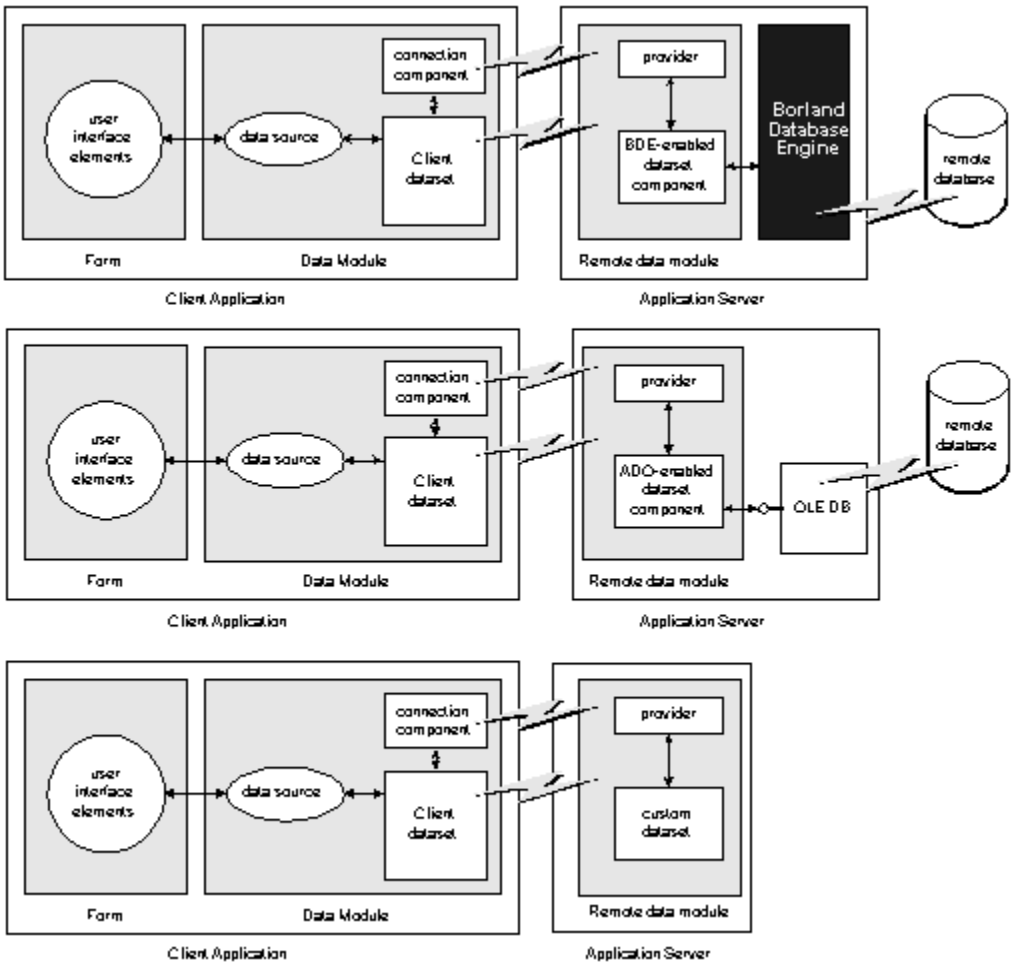
In this model, all applications are database clients. A *client* requests information from and sends information to a database server. A server can process requests from many clients simultaneously, coordinating access to and updating of data.

For more information on building two-tiered database applications, see BDE-based applications and ADO-based applications.

Understanding the architecture of a multi-tiered application

Topic groups See also

In multi-tiered database applications, an application is partitioned into pieces that reside on different machines. A client application provides a user interface to data. It passes all data requests and updates through an application server (also called a “remote data broker”). The application server, in turn, communicates directly with a remote database server or some other custom dataset. Usually, in this model, the client application, the application server, and the remote database server are on separate machines. The following figure illustrates these relationships for different types of multi-tiered applications.



You can use Delphi to create both client applications and application servers. The client application uses standard data-aware controls connected through a data source to one or more client dataset components in order to display data for viewing and editing. Each client dataset communicates with an application server through an *IAppServer* interface that is implemented by the application server’s remote data module. The client application can use a variety of protocols (TCP/IP, HTTP, DCOM, MTS, or CORBA) to establish this communication. The protocol depends on the type of connection component used in the client application and the type of remote data module used in the server application.

The application server contains provider components that mediate the communication between client datasets on the client application and the datasets on the application server. All data is passed between the client application and the provider components through the *IAppServer* interface.

Usually, several client applications communicate with a single application server in the multi-tiered model. The application server provides a gateway to your databases for all your client applications, and

it lets you provide enterprise-wide database tasks in a central location, accessible to all your clients. For more information about creating and using a multi-tiered database application, see Creating multi-tiered applications.

Designing the user interface

Topic groups See also

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

Data-aware controls get data from and send data to a data source component (*TDataSource*. A data source component acts as a conduit between the user interface and a dataset component which represents a set of information from the tables in a database. Several data-aware controls on a form can share a single data source, in which case the display in each control is synchronized so that as the user scrolls through records, the corresponding value in the fields for the current record is displayed in each control. An application's data source components usually reside in a data module, separate from the data-aware controls on forms.

The data-aware controls you add to your user interface depend on what type of data you are displaying (plain text, formatted text, graphics, multimedia elements, and so on). In addition, your choice of controls is determined by how you want to organize the information and how (or if) you want to let users navigate through the records of datasets and add or edit data.

The following sections introduce the components you can use for various types of user interface:

- Displaying a single record
- Displaying multiple records
- Analyzing data
- Selecting what data to show
- Writing reports

Displaying a single record

Topic groups See also

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

Applications that display a single record are usually easy to read and understand, because all database information is about the same thing (in the previous case, the same order). The data-aware controls in these user interfaces represent a single field from a database record. The Data Controls page of the Component palette provides a wide selection of controls to represent different kinds of fields. For more information about specific data-aware controls, see Controls that represent a single field.

Displaying multiple records

Topic groups See also

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

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

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

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

Tip: It is generally not a good idea to combine these two approaches on a single form. While the result can sometimes be effective, it is usually confusing for users to understand the data relationships.

Analyzing data

Topic groups See also

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

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

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

If you want to build your own components that display data summaries based on various grouping criteria, you can use maintained aggregates with a client dataset.

Selecting what data to show

Topic groups See also

Often, the data you want to surface in your database application does not correspond exactly to the data in a single database table. You may want to use only a subset of the fields or a subset of the records in a table. You may want to combine the information from more than one table into a single joined view.

The data available to your database application is controlled by your choice of dataset component. Datasets abstract the properties and methods of a database table, so that you do not need to make major alterations depending on whether the data is stored in a database table or derived from one or more tables in the database. For more information on the common properties and methods of datasets, see Understanding datasets.

Your application can contain more than one dataset. Each dataset represents a logical table. By using datasets, your application logic is buffered from restructuring of the physical tables in your databases. You might need to alter the type of dataset component, or the way it specifies the data it contains, but the rest of your user interface can continue to work without alteration.

When using the BDE to access your data, you can use any of the following types of dataset:

- **Table components (TTable):** Tables correspond directly to the underlying tables in the database. You can adjust which fields appear (including adding lookup fields and calculated fields) by using persistent field components. You can limit the records that appear using ranges or filters. Tables are described in more detail in Working with tables. Persistent fields are described in Persistent field components. Ranges and filters are described in Working with a subset of data.
- **Query components (TQuery):** Queries provide the most general mechanism for specifying what appears in a BDE-based dataset. You can combine the data from multiple tables using joins, and limit the fields and records that appear based on any criteria you can express in SQL. For more information on queries, see Working with queries.
- **Stored procedures (TStoredProc):** Stored procedures are sets of SQL statements that are named and stored on an SQL server. If your database server defines a stored procedure that returns the dataset you want, you can use a stored procedure component. For more information on stored procedures, see Working with stored procedures.
- **Nested datasets (TNestedTable):** Nested datasets represent the records in an Oracle8 nested detail set. Delphi does not let you create Oracle8 tables with nested dataset fields, but you can edit and display data from existing dataset fields using nested datasets. The nested dataset gets its data from a dataset field component in a dataset which contains Oracle8 data. See Working with nested tables and Working with dataset fields for more information on using nested datasets to represent dataset fields.

When using ADO to access your data, you can use any of the following types of dataset:

- **ADO datasets (TADODataset):** ADO datasets provide the most flexible mechanism for accessing data using ADO.ADO datasets can represent a single database table or the results of an SQL query. You can adjust which fields appear (including adding lookup fields and calculated fields) by using persistent field components. You can limit the records that appear using ranges or filters. You can specify an SQL statement that generates the data. ADO datasets are described in more detail in Features common to all ADO dataset components and Using TADODataSet.
- **ADO table components (TADOTable):** ADO tables correspond directly to the underlying tables in the database. You can adjust which fields appear (including adding lookup fields and calculated fields) by using persistent field components. You can limit the records that appear using ranges or filters. ADO tables are described in more detail in Using TADOTable.
- **ADO query components (TADOQuery):** ADO Queries represent the result set from running an SQL command or data definition language (DDL) statement. For more information on ADO queries, see Using TADOQuery.
- **ADO stored procedures (TADOStoredProc):** If your database server defines a stored procedure that returns the dataset you want, you can use an ADO stored procedure component. For more information on ADO stored procedures, see Using TADOStoredProc.

If you are using InterBase for your database server, you can use any of the following types of dataset:

- **IB datasets (TIBDataSet):** IB datasets represents the result set of an SQL statement (usually a SELECT statement). You can specify SQL statements for selecting and updating the data buffered by the

dataset.

- **IB table components (TIBTable):** IB tables get their data directly from an InterBase table or view. You can adjust which fields appear (including adding lookup fields and calculated fields) by using persistent field components. You can limit the records that appear using filters.
- **IB query components (TIBQuery):** IB queries represent the result set from running an SQL command. IB queries are the most easily scaled dataset component when moving from local InterBase to a remote InterBase server.
- **IB stored procedures (TIBStoredProc):** IBStoredProc executes an InterBase Execute stored procedure. These datasets do not return a result set: for stored procedures that return a result set you must use *TIBDataSet* or *TIBQuery*.

If you are not using the BDE, ADO, or InterBase, Delphi provides the following options:

- **Client datasets (TClientDataSet):** Client datasets cache the records of the logical dataset in memory. Because of that, they can only hold a limited number of records. Client datasets are populated with data in one of two ways: from an application server or from flat-file data stored on disk. They do not require a database engine such as BDE or ADO, but rely on a single DLL (Midas.dll). For more information about client datasets, see Creating and using a client dataset.
- **Custom datasets:** You can create your own custom descendants of *TDataSet* to represent a body of data that you create or access in code you write. Writing custom datasets allows you the flexibility of managing the data using any method you choose, while still letting you use the VCL data controls to build your user interface. For more information about creating custom components, see Overview of component creation.

Writing reports

Topic groups See also

If you want to let your users print database information from the datasets in your application, you can use the report components on the QReport page of the Component palette. Using these components you can visually build banded reports to present and summarize the information in your database tables. You can add summaries to group headers or footers to analyze the data based on grouping criteria.

Start a report for your application by selecting the QuickReport icon from the New Items dialog. Select File|New from the main menu, and go to the page labeled Business. Double-click the QuickReport Wizard icon to launch the wizard.

Note: See the QuickReport demo that ships with Delphi for an example of how to use the components on the QReport page.

InterBase Express

Topic groups

One- and two-tiered applications

Topic groups See also

One- and two-tiered applications include the logic that manipulates database information in the same application that implements the user interface. Because the data manipulation logic is not isolated in a separate tier, these types of applications are most appropriate when there are no other applications sharing the same database information. Even when other applications share the database information, these types of applications are appropriate if the database is very simple, and there are no data semantics that must duplicated by all applications that use the data.

You may want to start by writing a one- or two-tiered application, even when you intend to eventually scale up to a multi-tiered model as your needs increase. This approach lets you avoid having to develop data manipulation logic up front so that the application server can be available while you are writing the user interface. It also allows you to develop a simpler, cheaper prototype before investing in a large, multi-system development project. If you intend to eventually scale up to a multi-tiered application, you can isolate the data manipulation logic so that it is easy to move it to a middle tier at a later date.

Delphi provides support for two types of single-tiered applications: applications that use a local database (such as Paradox, dBase, Access, or Local InterBase) and flat-file database applications. Two-tiered applications use a driver to access a remote database.

The considerations when writing single-tired applications that use a local database and two-tiered applications are essentially the same, and depend primarily on the mechanism you choose to connect to the database. Delphi provides three different built-in mechanisms for these types of applications:

- BDE-based applications
- ADO-based applications
- InterBase Express applications

Flat file database applications are based on the support for client datasets included in MIDAS.DLL. For more information on these, see flat-file database applications.

BDE-based applications

Topic groups See also

Because the data access components (and Borland Database Engine) handle the details of reading data, updating data, and navigating data, writing BDE-based two-tiered applications is essentially the same as writing BDE-based one-tiered applications. See BDE-based architecture for more information about the structure of these applications.

When deploying BDE-based applications, you must include the BDE with your application. While this increases the size of the application and the complexity of deployment, the BDE can be shared with other BDE-based applications and provides many advantages. BDE-based applications allow you to use the powerful library of Borland Database Engine API calls. Even if you do not want to use the BDE API, writing BDE-based applications gives you support for the following features not available to other applications such as flat-file database application:

- Connecting to databases.
- Using transactions.
- Caching updates.
- Creating and restructuring database tables.

BDE-based architecture

Topic groups See also

A BDE-based one- or two-tiered application includes

- A user interface containing data-aware controls.
- One or more datasets that represent information from the database tables.
- A datasource component for each dataset to connect the data-aware controls to the datasets.
- Optionally, one or more database components to control transactions in both one- and two-tiered applications and to manage database connections in two-tiered applications.
- Optionally, one or more session components to isolate data access operations such as database connections, and to manage groups of databases.

The relationships between these elements is illustrated in the following figure:

The following topics provide additional information about these pieces:

- Designing the user interface
- Understanding databases and datasets
- Using data sources
- Using sessions

Understanding databases and datasets

Topic groups See also

Databases contain information stored in tables. They may also include tables of information about what is contained in the database, objects such as indexes that are used by tables, and SQL objects such as stored procedures. See Connecting to databases for more information about databases.

The Data Access page of the Component palette contains various dataset components that represent the tables contained in a database or logical tables constructed out of data stored in those database tables. See Selecting what data to show for more information about these dataset components. You must include a dataset component in your application to work with database information.

Each BDE-enabled dataset component on the Data Access page has a published *DatabaseName* property that specifies the database which contains the table or tables that hold the information in that dataset. When setting up your application, you must use this property to specify the database before you can bind the dataset to specific information contained in that database. What value you specify depends on whether

- The database has a BDE alias. You can specify a BDE alias as the value of *DatabaseName*. A BDE alias represents a database plus configuration information for that database. The configuration information associated with an alias differs by database type (Oracle, Sybase, InterBase, Paradox, dBASE, and so on). Use the BDE Administration tool or the SQL explorer to create and manage BDE aliases.
- The database is a Paradox or dBASE database. If you are using a Paradox or dBASE database, *DatabaseName* can specify the directory where the database tables are located.
- You are using explicit database components. Database components (*TDatabase*) represent a database in your application. If you don't add a database component explicitly, a temporary one is created for you automatically, based on the value of the *DatabaseName* property. If you are using explicit database components, *DatabaseName* is the value of the *DatabaseName* property of the database component. See Understanding persistent and temporary database components for more information about using database components.

Using sessions

Topic groups See also

Sessions isolate data access operations such as database connections, and manage groups of databases. All use of the Borland Database Engine takes place in the context of a session. You can use sessions to specify configuration information that applies to all the databases in the session. This allows you to override the default behavior specified using the BDE administration tool.

You can use a session to

- Manage BDE aliases. You can create new aliases, delete aliases, and modify existing aliases. By default, changes affect only the session, but you can write changes so that they are added to the permanent BDE configuration file. For more information on managing BDE aliases, see Working with BDE aliases.
- Control when database connections in two-tiered applications are closed. Keeping database connections open when none of the datasets in the database are active ties up resources that could be released, but improves speed and reduces network traffic. To keep database connections open even when there are no active datasets, the *KeepConnections* property should be *True*(the default).
- Manage access to password-protected Paradox and dBASE files in one-tiered applications. Datasets that access password-protected Paradox and dBASE tables use the session component to supply a password when these tables must be opened. You can override the default behavior (a password dialog that appears whenever a password is needed), to supply passwords programmatically. If you intend to scale your one-tiered application to a two-tiered or multi-tiered application, you can create a common user interface for obtaining user authentication information that need not change when you switch to using remote database servers which require a username and password at the server (rather than table) level. For more information about using sessions to manage Paradox and dBASE passwords, see Working with password-protected Paradox and dBase tables.
- Specify the location of special Paradox directories. Paradox databases that are shared on a network use a net directory which contains temporary files that specify table and record locking information. Paradox databases also use a private directory where temporary files such as the results of queries are kept. For more information on specifying these directory locations, see Specifying Paradox directory locations.

If your application may be accessing the same database multiple times simultaneously, you must use multiple sessions to isolate these uses of the database. Failure to do so will disrupt the logic governing transactions on that database (including transactions created for you automatically). Applications risk simultaneous access when running concurrent queries or when using multiple threads. For more information about using multiple sessions, see Managing multiple sessions.

Unless you need to use multiple sessions, you can use the default session.

Connecting to databases

Topic groups See also

The Borland Database Engine includes drivers to connect to different databases. The Standard version of Delphi includes only the drivers for local databases: Paradox, dBASE, FoxPro, and Access. With the Professional version, you also get an ODBC adapter that allows the BDE to use ODBC drivers. By supplying an ODBC driver, your application can use any ODBC-compliant database. Some versions also include drivers for remote database servers. Use the drivers installed with SQL Links to communicate with remote database servers such as InterBase, Oracle, Sybase, Informix, Microsoft SQL server, and DB2.

Note: The only difference between a BDE-based one-tiered application and a BDE-based two-tiered application is whether it uses local databases or remote database servers.

Using transactions

Topic groups See also

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

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

By default, the BDE provides implicit transaction control for your applications. When an application is under implicit transaction control, a separate transaction is used for each record in a dataset that is written to the underlying database. Implicit transactions guarantee both a minimum of record update conflicts and a consistent view of the database. On the other hand, because each row of data written to a database takes place in its own transaction, implicit transaction control can lead to excessive network traffic and slower application performance. Also, implicit transaction control will not protect logical operations that span more than one record, such as the transfer of funds described previously.

If you explicitly control transactions, you can choose the most effective times to start, commit, and roll back your transactions. When you develop applications in a multi-user environment, particularly when your applications run against a remote SQL server, you should control transactions explicitly. For more information about explicitly controlling transactions, see Explicitly controlling transactions.

Note: You can also minimize the number of transactions you need by caching updates. For more information about cached updates, see Working with cached updates.

Explicitly controlling transactions

Topic groups See also

There are two mutually exclusive ways to control transactions explicitly in a BDE-based database application:

- Use the methods and properties of the database component, such as *StartTransaction*, *Commit*, *Rollback*, *InTransaction*, and *TransIsolation*. The main advantage to using the methods and properties of a database component to control transactions is that it provides a clean, portable application that is not dependent on a particular database or server.
- Use passthrough SQL in a query component to pass SQL statements directly to remote SQL or ODBC servers. For more information about query components, see "Working with queries." The main advantage to passthrough SQL is that you can use the advanced transaction management capabilities of a particular database server, such as schema caching. To understand the advantages of your server's transaction management model, see your database server documentation.

One-tiered applications can't use passthrough SQL. You can use the database component to create explicit transactions for local databases. However, there are limitations to using local transactions. For more information on using local transactions, see Using local transactions.

When writing two-tiered applications (which require SQL links), you can use either a database component or passthrough SQL to manage transactions.

Using a database component for transactions

Topic groups See also

When you start a transaction, all subsequent statements that read from and write to the database occur in the context of that transaction. Each statement is considered part of a group. Changes must be successfully committed to the database, or every change made in the group must be undone.

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

When using a database component, you code a single transaction as follows:

1. Start the transaction by calling the database's *StartTransaction* method.
2. Once the transaction is started, all subsequent database actions are considered part of the transaction until the transaction is explicitly terminated. You can determine whether a transaction is in process by checking the database component's *InTransaction* property. While the transaction is in process, your view of the data in database tables is determined by you transaction isolation level. For more information about transaction isolation levels, see Using the TransIsolation property.
3. When the actions that make up the transaction have all succeeded, you can make the database changes permanent by using the database component's *Commit* method.

   Commit is usually attempted in a **try...except** statement. That way, if a transaction cannot commit successfully, you can use the **except** block to handle the error and retry the operation or to roll back the transaction.
4. If an error occurs when making the changes that are part of the transaction, or when trying to commit the transaction, you will want to discard all changes that make up the transaction. To discard these changes, use the database component's *Rollback* method.

   Rollback usually occurs in

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

Using the TransIsolation property

Topic groups See also

TransIsolation specifies the *transaction isolation level* for a database component's transactions. Transaction isolation level determines how a transaction interacts with other simultaneous transactions when they work with the same tables. In particular, it affects how much a transaction "sees" of other transactions' changes to a table.

The default setting for *TransIsolation* is *tiReadCommitted*. The following table summarizes possible values for *TransIsolation* and describes what they mean:

Isolation level	Meaning
tiDirtyRead	Permit reading of uncommitted changes made to the database by other simultaneous transactions. Uncommitted changes are not permanent, and might be rolled back (undone) at any time. At this level your transaction is least isolated from the changes made by other transactions.
tiReadCommitted	Permit reading only of committed (permanent) changes made to the database by other simultaneous transactions. This is the default isolation level.
tiRepeatableRead	Permit a single, one time reading of the database. Your transaction cannot see any subsequent changes to data by other simultaneous transactions. This isolation level guarantees that once your transaction reads a record, its view of that record will not change. At this level your transaction is most isolated from changes made by other transactions.

Database servers may support these isolation levels differently or not at all. If the requested isolation level is not supported by the server, the BDE uses the next highest isolation level. The actual isolation level used by some servers is shown in the following table. For a detailed description of how each isolation level is implemented, see your server documentation.

Server	Specified Level	Actual Level
Oracle	tiDirtyRead	tiReadCommitted
	tiReadCommitted	tiReadCommitted
	tiRepeatableRead	tiRepeatableRead (READONLY)
Sybase, MS-SQL	tiDirtyRead	tiReadCommitted
	tiReadCommitted	tiReadCommitted
	tiRepeatableRead	Not supported
DB2	tiDirtyRead	tiDirtyRead
	tiReadCommitted	tiReadCommitted
	tiRepeatableRead	tiRepeatableRead
Informix	tiDirtyRead	tiDirtyRead
	tiReadCommitted	tiReadCommitted
	tiRepeatableRead	tiRepeatableRead
InterBase	tiDirtyRead	tiReadCommitted
	tiReadCommitted	tiReadCommitted
	tiRepeatableRead	tiRepeatableRead
Paradox, dBASE, Access, FoxPro	tiDirtyRead	tiDirtyRead
	tiReadCommitted	Not supported
	tiRepeatableRead	Not supported

Note: When using transactions with local Paradox, dBASE, Access, and FoxPro tables, set *TransIsolation* to *tiDirtyRead* instead of using the default value of *tiReadCommitted*. A BDE error is returned if *TransIsolation* is set to anything but *tiDirtyRead* for local tables.

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 passthrough SQL

Topic groups See also

With passthrough SQL, you use a *TQuery*, *TStoredProc*, or *TUpdateSQL* component to send an SQL transaction control statement directly to a remote database server. The BDE does not process the SQL statement. Using passthrough SQL enables you to take direct advantage of the transaction controls offered by your server, especially when those controls are non-standard.

To use passthrough SQL to control a transaction, you must

- Install the proper SQL Links drivers. If you chose the “Typical” installation when installing Delphi, all SQL Links drivers are already properly installed.
- Configure your network protocol correctly. See your network administrator for more information.
- Have access to a database on a remote server.
- Set SQLPASSTHRU MODE to NOT SHARED using the SQL Explorer. SQLPASSTHRU MODE specifies whether the BDE and passthrough SQL statements can share the same database connections. In most cases, SQLPASSTHRU MODE is set to SHARED AUTOCOMMIT. However, you can’t share database connections when using transaction control statements. For more information about SQLPASSTHRU modes, see the help file for the BDE Administration utility.

Note: When SQLPASSTHRU MODE is NOT SHARED, you must use separate database components for datasets that pass SQL transaction statements to the server and datasets that do not.

Using local transactions

Topic groups See also

The BDE supports local transactions against local Paradox, dBASE, Access, and FoxPro tables. From a coding perspective, there is no difference to you between a local transaction and a transaction against a remote database server.

When a transaction is started against a local table, updates performed against the table are logged. Each log record contains the old record buffer for a record. When a transaction is active, records that are updated are locked until the transaction is committed or rolled back. On rollback, old record buffers are applied against updated records to restore them to their pre-update states.

Local transactions are more limited than transactions against SQL servers or ODBC drivers. In particular, the following limitations apply to local transactions:

- Automatic crash recovery is not provided.
- Data definition statements are not supported.
- Transactions cannot be run against temporary tables.
- For Paradox, local transactions can only be performed on tables with valid indexes. Data cannot be rolled back on Paradox tables that do not have indexes.
- Only a limited number of records can be locked and modified. With Paradox tables, you are limited to 255 records. With dBASE the limit is 100.
- Transactions cannot be run against the BDE ASCII driver.
- *TransIsolation* level must only be set to *tiDirtyRead*.
- Closing a cursor on a table during a transaction rolls back the transaction unless:
- Several tables are open.
- The cursor is closed on a table to which no changes were made.

Caching updates

Topic groups See also

The Borland Database Engine provides support for caching updates. When you cache updates, your application retrieves data from a database, makes all changes to a local, cached copy of the data, and applies the cached changes to the dataset as a unit. Cached updates are applied to the database in a single transaction.

Caching updates can minimize transaction times and reduce network traffic. However, cached data is local to your application and is not under transaction control. This means that while you are working on your local, in-memory, copy of the data, other applications can be changing the data in the underlying database table. They also can't see any changes you make until you apply the cached updates. Because of this, cached updates may not be appropriate for applications that work with volatile data, as you may create or encounter too many conflicts when trying to merge your changes into the database.

You can tell BDE-enabled datasets to cache updates using the *CachedUpdates* property. When the changes are complete, they can be applied by the dataset component, by the database component, or by a special update object. When changes can't be applied to the database without additional processing (for example, when working with a joined query), you must use the *OnUpdateRecord* event to write changes to each table that makes up the joined view.

For more information on caching updates, see Working with cached updates.

Note: If you are caching updates, you may want to consider moving to a multitiered model to have greater control over the application of updates. For more information about the multitiered model, see Creating multi-tiered applications.

Creating and restructuring database tables

Topic groups See also

In BDE-based applications, you can use the *TTable* component to create new database tables and to add indexes to existing tables.

You can create tables either at design time, in the Forms Designer, or at runtime. To create a table, you must specify the fields in the table using the *FieldDefs* property, add any indexes using the *IndexDefs* property, and call the *CreateTable* method (or select the Create Table command from the table's context menu). For more detailed instructions on creating tables, see Creating a table.

Note: When creating Oracle8 tables, you can't create object fields (ADT fields, array fields, reference fields, and dataset fields).

If you want to restructure a table at runtime (other than by adding indexes), you must use the BDE API *DbiDoRestructure*. You can add indexes to an existing table using the *AddIndex* method of *TTable*.

Note: At design time, you can use the Database Desktop to create and restructure Paradox and dBASE tables. To create and restructure tables on remote servers, use the SQL Explorer and restructure the table using SQL.

ADO-based applications

Topic groups See also

Delphi applications that use the ADO (ActiveX Data Objects) components for data access can be either one- or two-tier. Which category an application falls under is predicated on the database type used. For instance, using ADO to access a Microsoft SQL Server database will always be a two-tier application because SQL Server is an SQL database system. SQL database systems are typically located on a dedicated SQL server. On the other hand, an application that uses ADO to access some local database type, like dBASE or FoxPro, will always be a one-tier application.

There are four major areas involved in database access using ADO and the Delphi ADO components. These areas of concern are the same regardless of whether the application is one- or two-tier. These five concerns are:

- ADO-based architecture
- Connecting to ADO databases
- Retrieving data
- Creating and restructuring ADO database tables

ADO-based architecture

Topic groups

An ADO-based application includes the following functional areas

- A user interface with visual data-aware controls. Visual data controls are optional if all data access is done programmatically.
- One or more dataset components that represent information from tables or queries.
- One datasource component for each dataset component to act as the conduit between dataset component and one or more visual data-aware controls.
- A connection component to connect to the ADO data store. The connection component acts as a conduit between the application's dataset components and the database accessed through the data store.

The ADO layer of an ADO-based Delphi application consists of Microsoft ADO 2.1, an OLE DB provider or ODBC driver for the data store access, client software for the specific database system used (in the case of SQL databases), a database back-end system accessible to the application (for SQL database systems), and a database. All of these external entities must be present and accessible to the ADO-based application for it to be fully functional.

Understanding ADO databases and datasets

Topic groups See also

The ADO (ActiveX Data Objects) page of the Component Palette contains all of the components necessary for connecting to databases and for accessing the tables in them.

All of the metadata objects an ADO-based application are contained in the database accessed through the ADO data store. To access these objects or the data stored in them, an application must first connect to the data store. See Connecting to ADO data stores for information on connecting to data stores.

The data of a database is stored in one or more tables. You must include at least one ADO dataset component (*TADODataSet*, *TADOQuery*, and so on) in your application to work with the data stored in a database's tables. See Retrieving data and Using ADO datasets for more information on using ADO dataset components to access data in tables.

Connecting to ADO databases

Topic groups

An ADO-based application Delphi uses Microsoft ActiveX Data Objects (ADO) 2.1 to interact with an OLE DB provider to connect to a data store and access its data. One of the things a data store can represent is a database. An ADO-based application requires that ADO 2.1 be installed on the client computer. ADO and OLE DB is supplied by Microsoft and installed with Windows.

The provider can represent one of a number of types of access, from native OLE DB drivers to ODBC drivers. These drivers must also be installed on the client computer. OLE DB drivers for various database systems are supplied by the database vendor or by a third-party.

If the application uses an SQL database, such as Microsoft SQL Server or Oracle, the client software for that database system must also be installed on the client computer. Client software is supplied by the database vendor and installed from the database systems CD (or disk).

To connect the application with the data store, the ADO connection component is configured to use one of the available providers. Once the connection component is connected to the data store, dataset components can be associated with the connection component to access the tables in the database. See Connecting to ADO data stores for information on connecting to data stores.

In addition to providing an application's access to the database, the connection component encapsulates the ADO transaction processing capabilities.

Retrieving data

Topic groups

Once an application has established a valid connection to a database, dataset components can be used to access data in the database. The ADO page of the Component Palette contains the ADO dataset components needed to access data from ADO data stores.

These components include *TADODataSet*, *TADOTable*, *TADOQuery*, and *TADOStoredProc*. All of these components are capable of retrieving data from ADO data stores, programmatically modifying the data, and presenting the data to an application's user for interactive use of the data. For more information on using the ADO dataset components to retrieve and modify data, see Using ADO datasets.

To make the data accessed with an ADO dataset component visually accessible in an application, use the stock data-aware controls. There are no ADO-specific data-aware controls. For details on using data-aware controls, see Using common data control features.

The standard data source component is used as a conduit between the ADO dataset components and the data-aware controls. There is no dedicated data source component for ADO. For details on using data source components, see Using data sources.

Where needed, persistent field objects can be used to represent fields in the ADO dataset components. As with the data-aware controls and data source components, simply use the inherent Delphi field classes (*TField* and descendants). For details on using dynamic and persistent field objects, see Understanding field components.

Creating and restructuring ADO database tables

Topic groups See also

Creating and deleting metadata in an ADO database from a Delphi application must be done using SQL. Similarly, restructuring tables is done using SQL statements. Changing other metadata objects cannot be done per se. Instead, you need to delete the metadata object and then replace it with a new one with different attributes.

There are many database types that can be accessed through ADO and not all of the drivers for specific database types support all of the same SQL syntax. It is beyond the scope of this document to describe all of the SQL syntax supported by each database type and all of the differences between the database types. For a comprehensive and up-to-date discussion of the SQL implementation for a given database system, see the documentation that comes with that database system.

In general, use the CREATE TABLE statement to create tables in the database and CREATE INDEX to create new indexes for those tables. Where supported, use other CREATE statements for adding various metadata objects, such as CREATE DOMAIN, CREATE VIEW, and CREATE SCHEMA.

For each of the CREATE statements, there is a corresponding DROP statement to delete a metadata object. These statements include DROP TABLE, DROP VIEW, DROP DOMAIN, and DROP SCHEMA.

To change the structure of a table, use the ALTER TABLE statement. ALTER TABLE has ADD and DROP clauses to create new elements in a table and to delete them. For example, use the ADD COLUMN clause to add a new column to the table and DROP CONSTRAINT to delete an existing constraint from the table.

Issue these metadata statements from the ADO command or the ADO query component. For details on using the ADO command component to execute commands, see Executing commands.

Flat-file database applications

Topic groups See also

Flat-file database applications are single-tiered applications that use *TClientDataSet* to represent all of their datasets. The client dataset holds all its data in memory, which means that this type of application is not appropriate for extremely large datasets.

Flat-file database applications do not require the Borland Database Engine (BDE) or ActiveX Data Objects (ADO). Instead, they only use MIDAS.DLL. By using only MIDAS.DLL, flat-file applications are easier to deploy because you do not need to install, configure, and maintain software that manages database connections.

Because these applications do not use a database, there is no support for multiple users. Instead, the datasets are dedicated entirely to the application. Data can be saved to flat files on disk, and loaded at a later time, but there is no built-in protection to prevent multiple users from overwriting each other's data files.

Client datasets (located on the MIDAS page of the Component palette) form the basis of flat-file database applications. They provide support for most of the database operations you perform with other datasets. You use the same data-aware controls and data source components that you would use in a BDE-based single-tiered application. You don't use database components, because there is no database connection to manage, and no transactions to support. You do not need to be concerned with session components unless your application is multi-threaded. For more information about using client datasets, see Creating and using a client dataset.

The main differences in writing flat-file database applications and other single-tiered database applications lie in how you create the datasets and how you load and save data.

You can create hybrid applications that act as single-tiered flat-file database applications, and, when connected to an application server, act as the client portion of a multi-tiered database application. This is called using the briefcase model

Creating the datasets

Topic groups See also

Because flat-file database applications do not use existing databases, you are responsible for creating the datasets yourself. Once the dataset is created, you can save it to a file. From then on, you do not need to recreate the table, only load it from the file you saved. However, indexes are not saved with the table. You need to recreate them every time you load the table.

When beginning a flat-file database application, you may want to first create and save empty files for your datasets before beginning the writing of the application itself. This way, you do not need to define the metadata for your client datasets in the final application.

How you create your client dataset depends on whether you are creating an entirely new dataset, or converting an existing BDE-based application.

- See Creating a new dataset using persistent fields for information on how to create this dataset using the Fields editor.
- See Creating a dataset using field and index definitions for information on how to create this using the FieldDefs and IndexDefs properties.
- See Creating a dataset based on an existing table for information on how to copy an existing table to create a new client dataset.

Creating a new dataset using persistent fields

Topic groups See also

The following steps describe how to create a new client dataset using the Fields Editor:

1. From the MIDAS page of the Component palette, add a *TClientDataSet* component to your application.
2. Right-click the client dataset and select Fields Editor. In the Fields editor, right-click and choose the New Field command. Describe the basic properties of your field definition. Once the field is created, you can alter its properties in the Object Inspector by selecting the field in the Fields editor.

   Continue adding fields in the fields editor until you have described your client dataset.
3. Right-click the client dataset and choose Create DataSet. This creates an empty client dataset from the persistent fields you added in the Fields Editor.
4. Right-click the client dataset and choose Save To File. (This command is not available unless the client dataset contains data.)
5. In the File Save dialog, choose a file name and save a flat file copy of your client dataset.

Note: You can also create the client dataset at runtime using persistent fields that are saved with the client dataset. Simply call the *CreateDataSet* method.

Creating a dataset using field and index definitions

Topic groups See also

Creating a client dataset using field and index definitions is much like using a *TTable* component to create a database table. There is no *DatabaseName*, *TableName*, or *TableType* property to specify, as these are not relevant to client datasets. However, just as with *TTable*, you use the *FieldDefs* property to specify the fields in your table and the *IndexDefs* property to specify any indexes. Once the table is specified, right-click the client dataset and choose Create DataSet at design time, or call the *CreateDataSet* method at runtime.

When defining the index definitions for your client dataset, two properties of the index definition apply uniquely to client datasets. These are *TIndexDef.DescFields* and *TIndexDef.CaseInsFields*.

DescFields lets you define indexes that sort records in ascending order on some fields and descending order on other fields. Instead of using the *ixDescending* option to sort in descending order on all the fields in the index, list only those fields that should sort in descending order as the value of *DescFields*. For example, when defining an index that sorts on Field1, then Field2, then Field3, setting *DescFields* to

```
Field1;Field3
```

results in an index that sorts Field2 in ascending order and Field1 and Field3 in descending order.

CaseInsFields lets you define indexes that sort records case-sensitively on some fields and case-insensitively on other fields. Instead of using the *isCaseInsensitive* option to sort case-insensitively on all the fields in the index, list only those fields that should sort case-insensitively as the value of *CaseInsFields*. Like *DescFields*, *CaseInsFields* takes a semicolon-delimited list of field names.

You can specify the field and index definitions at design time using the Collection editor. Just choose the appropriate property in the Object Inspector (*FieldDefs* or *IndexDefs*), and double-click to display the Collection editor. Use the Collection editor to add, delete, and rearrange definitions. By selecting definitions in the Collection editor you can edit their properties in the Object Inspector.

You can also specify the field and index definitions in code at runtime. For example, the following code creates and activates a client dataset in the form's *OnCreate* event handler:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
with ClientDataSet1 do
begin
  with FieldDefs.AddFieldDef do
  begin
    DataType := ftInteger;
    Name := 'Field1';
  end;
  with FieldDefs.AddFieldDef do
  begin
    DataType := ftString;
    Size := 10;
    Name := 'Field2';
  end;
  with IndexDefs.AddIndexDef do
  begin
    Fields := 'Field1';
    Name := 'IntIndex';
  end;
  CreateDataSet;
end;
end;
```

Creating a dataset based on an existing table

Topic groups See also

If you are converting an existing BDE-based application into a single-tiered flat-file application, you can copy existing tables and save them as flat-file tables from the IDE. The following steps indicate how to copy an existing table:

1. From the Data Access page of the Component palette, add a *TTable* component to your application. Set its *DatabaseName* and *TableName* properties to identify the existing database table. Set its *Active* property to *True*.
2. From the MIDAS page of the Component palette, add a *TClientDataSet* component.
3. Right-click the client dataset and select Assign Local Data. In the dialog that appears, choose the table component that you added in step 1. Choose OK.
4. Right-click the client dataset and choose Save To File. (This command is not available unless the client dataset contains data.)
5. In the File Save dialog, choose a file name and save a flat-file copy of your database table.

Loading and saving data

Topic groups See also

In flat-file database applications, all modifications to the table exist only in an in-memory change log. This log is maintained separately from the data itself, although it is completely transparent to objects that use the client dataset. That is, controls that navigate the client dataset or display its data see a view of the data that includes the changes. If you do not want to back out of changes, however, you should merge the change log into the data of the client dataset by calling the *MergeChangeLog* method. For more information about the change log, see Editing data.

Even when you have merged changes into the data of the client dataset, this data still exists only in memory. While it will persist if you close the client dataset and reopen it in your application, it will disappear when your application shuts down. To make the data permanent, it must be written to disk. Write changes to disk using the *SaveToFile* method. *SaveToFile* takes one parameter, the name of the file which is created (or overwritten) containing the table.

When you want to read a table previously written using the *SaveToFile* method, use the *LoadFromFile* method. *LoadFromFile* also takes one parameter, the name of the file containing the table.

When you save a client dataset, the metadata that describes the record structure is saved with the dataset, but not the indexes. Because of this, you may want to add code that recreates the indexes when you load the data from file. Alternately, you might want to write your application so that it always creates indexes on the fly in an as-needed fashion.

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

Supporting the briefcase model

Topic groups See also

The one-tiered model can be combined with a multi-tiered model to create what is called the briefcase model.

Note: The briefcase model is sometimes called the disconnected model, or mobile computing.

When operating on site, a briefcase model application looks like a multi-tiered model: a user starts a client application on one machine and connects over a network to an application server on a remote machine. The client requests data from the application server, and sends updates to it. The updates are applied by the application server to a database that is presumably shared with other clients throughout an organization.

Suppose, however, that your onsite company database contains valuable customer contact data that your sales representatives can use and update in the field. In this case, it would be useful if your sales reps could download some or all of the data from the company database, work with it on their laptops as they fly across the country, and even update records at existing or new customer sites. When the sales reps return onsite, they need to upload their data changes to the company database for everyone to use. This ability to work with data off-line and then apply updates online at a later date is known as the "briefcase" model.

By using the briefcase model, you can take advantage of the client dataset component's ability to read and write data to flat files to create client applications that can be used both online with an application server, and off-line, as temporary one-tiered applications.

To implement the briefcase model, you must

1. Create a multi-tiered server application as described in Creating the application server.
2. Create a flat-file database application as your client application. Add a connection component and set the *RemoteServer* property of your client datasets to specify this connection component. This allows them to talk to the application server created in step 1. For more information about connection components, see Connecting to the application server.
3. In the client application, try on start-up to connect to the application server. If the connection fails, prompt the user for a file and read in the local copy of the data.
4. In the client application, add code to apply updates to the application server. For more information on sending updates from a client application to an application server, see Updating records.

Scaling up to a three-tiered application

Topic groups See also

In a two-tiered client/server application, the application is a client that talks directly to a database server. Even so, the application can be thought of as having two parts: a database connection and a user interface. To make a two-tiered client/server application into a multi-tiered application you must:

- Split your existing application into an application server that handles the database connection, and a client application that contains the user interface.
- Add an interface between the client and the application server.

There are a number of ways to proceed, but the following sequential steps may best keep your translation work to a minimum:

1. Create a new project for the application server, starting with a remote data module. See Creating the application server for details on how to do this.
2. Duplicate the relevant database connection portions of your former two-tiered application, and for each dataset, add a provider component that will act as a data conduit between the application server and the client. For more information on using a provider component, see Using provider components.
3. Copy your existing two-tiered project, remove its direct database connections, add an appropriate connection component to it. For more information about creating and using connection components, see Connecting to the application server.
4. Substitute a client dataset for each dataset component in the original project. For general information about using a client dataset component, see Creating and using a client dataset.
5. In the client application, add code to apply updates to the application server. For more information on sending updates from a client application to an application server, see Updating records.
6. Move the dataset components to the application server's data modules. Set the *DataSet* property of each provider to specify the corresponding datasets. For more information about linking a dataset to a provider component, see Using provider components.

Creating multi-tiered applications

Topic groups See also

A multi-tiered client/server application is partitioned into logical units which run in conjunction on separate machines. Multi-tiered applications share data and communicate with one another over a local-area network or even over the Internet. They provide many benefits, such as centralized business logic and thin client applications.

In its simplest form, sometimes called the “three-tiered model,” a multi-tiered application is partitioned into thirds:

- **Client application**: provides a user interface on the user’s machine.
- **Application server**: resides in a central networking location accessible to all clients and provides common data services.
- **Remote database server**: provides the relational database management system (RDBMS).

In this three-tiered model, the application server manages the flow of data between clients and the remote database server, so it is sometimes called a “data broker.” With Delphi you usually only create the application server and its clients, although, if you are really ambitious, you could create your own database back end as well.

In more complex multi-tiered applications, additional services reside between a client and a remote database server. For example, there might be a security services broker to handle secure Internet transactions, or bridge services to handle sharing of data with databases on platforms not directly supported by Delphi.

Delphi support for multi-tiered applications is based on the Multi-tier Distributed Application Services Suite (MIDAS). See Understanding MIDAS technology for an overview of this technology and the architecture of a three-tiered application built using MIDAS. Once you understand how to create and manage a three-tiered application, you can create and add additional service layers based on your needs. Building a multi-tiered application provides details on how to apply this architecture to build a MIDAS-based application.

Advantages of the multi-tiered database model

Topic groups See also

The multi-tiered database model breaks a database application into logical pieces. The client application can focus on data display and user interactions. Ideally, it knows nothing about how the data is stored or maintained. The application server (middle tier) coordinates and processes requests and updates from multiple clients. It handles all the details of defining datasets and interacting with the remote database server.

The advantages of this multi-tiered model include the following:

- **Encapsulation of business logic in a shared middle tier**. Different client applications all access the same middle tier. This allows you to avoid the redundancy (and maintenance cost) of duplicating your business rules for each separate client application.
- **Thin client applications**. Your client applications can be written to make a small footprint by delegating more of the processing to middle tiers. Not only are client applications smaller, but they are easier to deploy because they don't need to worry about installing, configuring, and maintaining the database connectivity software (such as the Borland Database Engine). Thin client applications can be distributed over the internet for additional flexibility.
- **Distributed data processing**. Distributing the work of an application over several machines can improve performance because of load balancing, and allow redundant systems to take over when a server goes down.
- **Increased opportunity for security**. You can isolate sensitive functionality into tiers that have different access restrictions. This provides flexible and configurable levels of security. Middle tiers can limit the entry points to sensitive material, allowing you to control access more easily. If you are using HTTP, CORBA, or MTS, you can take advantage of the security models they support.

Understanding MIDAS technology

Topic groups See also

MIDAS provides the mechanism by which client applications and application servers communicate database information. Using MIDAS requires MIDAS.DLL, which is used by both client and server applications to manage datasets stored as data packets. Building MIDAS applications may also require the SQL explorer to help in database administration and to import server constraints into the Data Dictionary so that they can be checked at any level of the multi-tiered application.

Note: You must purchase server licenses for deploying your MIDAS applications.

MIDAS-based multi-tiered applications use the components on the MIDAS page of the component palette, plus a remote data module that is created by a wizard on the Multitier page of the New Items dialog. These components are described in the following table:

Component	Description
remote data modules	Specialized data modules that can act as a COM Automation server or CORBA server to give client applications access to any providers they contain. Used on the application server.
provider component	A data broker that provides data by creating data packets and resolves client updates. Used on the application server.
client dataset component	A specialized dataset that uses MIDAS.DLL to manage data stored in data packets.
connection components	A family of components that locate the server, form connections, and make the *IAppServer* interface available to client datasets. Each connection component is specialized to use a particular communications protocol.

For more information on how these components fit together to create a multi-tiered application, see

- Overview of a MIDAS-based multi-tiered application
- The structure of the client application
- The structure of the application server
- Choosing a connection protocol

Overview of a MIDAS-based multi-tiered application

Topic groups See also

The following numbered steps illustrate a normal sequence of events for a MIDAS-based multi-tiered application:

1. A user starts the client application. The client connects to the application server (which can be specified at design time or runtime). If the application server is not already running, it starts. The client receives an *IAppServer* interface from the application server.
2. The client requests data from the application server. A client may request all data at once, or may request chunks of data throughout the session (fetch on demand).
3. The application server retrieves the data (first establishing a database connection, if necessary), packages it for the client, and returns a data packet to the client. Additional information, (for example, about data constraints imposed by the database) can be included in the metadata of the data packet. This process of packaging data into data packets is called “providing.”
4. The client decodes the data packet and displays the data to the user.
5. As the user interacts with the client application, the data is updated (records are added, deleted, or modified). These modifications are stored in a change log by the client.
6. Eventually the client applies its updates to the application server, usually in response to a user action. To apply updates, the client packages its change log and sends it as a data packet to the server.
7. The application server decodes the package and posts updates in the context of a transaction. If a record can’t be posted to the server (for example, because another application changed the record after the client requested it and before the client applied its updates), the application server either attempts to reconcile the client’s changes with the current data, or saves the records that could not be posted. This process of posting records and caching problem records is called “resolving.”
8. When the application server finishes the resolving process, it returns any unposted records to the client for further resolution.
9. The client reconciles unresolved records. There are many ways a client can reconcile unresolved records. Typically the client attempts to correct the situation that prevented records from being posted or discards the changes. If the error situation can be rectified, the client applies updates again.
10. The client refreshes its data from the server.

The structure of the client application

Topic groups See also

To the end user, the client application of a multi-tiered application looks and behaves no differently than a traditional two-tiered application that uses cached updates. Structurally, the client application looks a lot like a flat-file single-tiered application. User interaction takes place through standard data-aware controls that display data from a client dataset component. For detailed information about using the properties, events, and methods of client datasets, see Creating and using a client dataset.

Unlike in a flat-file application, the client dataset in a multi-tiered application obtains its data through the *IAppServer* interface on the application server. It uses this interface to post updates to the application server as well. For more information about the *IAppServer* interface, see Using the IAppServer interface. The client gets this interface from a connection component.

The connection component establishes the connection to the application server. Different connection components are available for using different communications protocols. These connection components are summarized in the following table:

Component	Protocol
TDCOMConnection	DCOM
TSocketConnection	Windows sockets (TCP/IP)
TWebConnection	HTTP
TOLEnterpriseConnection	OLEnterprise (RPCs)
TCorbaConnection	CORBA (IIOP)

Note: Two other connection components, *TRemoteServer* and *TMIDASConnection,* are provided for backward compatibility.

For more information about using connection components, see Connecting to the application server.

The structure of the application server

Topic groups See also

The application server includes a remote data module that provides an *IAppServer* interface, which client applications use to communicate with data providers. There are three types of remote data modules:

- **TRemoteDataModule**: This is a dual-interface Automation server. Use this type of remote data module if clients use DCOM, HTTP, sockets, or OLEnterprise to connect to the application server, unless you want to install the application server with MTS.
- **TMTSDataModule**: This is a dual-interface Automation server. Use this type of remote data module if you are creating the application server as an Active Library (.DLL) that is installed with MTS. You can use MTS remote data modules with DCOM, HTTP, Sockets, or OLEnterprise. See Using MTS for information about the benefits and limitations of using MTS with the application server.
- **TCorbaDataModule**: This is a CORBA server. Use this type of remote data module to provide data to CORBA clients.

As with any data module, you can include any nonvisual component in the remote data module. In addition, the remote data module includes a dataset provider component for each dataset the application server makes available to client applications. A dataset provider

- Receives data requests from the client, fetches the requested data from the database server, packages the data for transmission, and sends the data to the client dataset. This activity is called “providing.”
- Receives updated data from the client dataset, applies updates to the database or source dataset, and logs any updates that cannot be applied, returning unresolved updates to the client for further reconciliation. This activity is called “resolving.”

Often, the provider uses BDE- or ADO-enabled datasets such as you find in a two-tiered application. You can add database and session components as needed, just as in a BDE-based two-tiered application, or ADO connection components as in an ADO-based two-tiered application.

Note: Do not confuse the ADO connection component, which is analogous to a database component in a BDE-based application, with the connection components used by client applications in a multitiered application.

For more information about two-tiered applications, see Building one- and two-tiered applications.

If the application server is MTS-enabled, the MTS data module includes events for when the application server is activated or deactivated. This permits the application server to acquire database connections when activated and release them when deactivated.

Using MTS

Topic groups See also

Using MTS lets your remote data module take advantage of

- **MTS security.** MTS provides role-based security for your application server. Clients are assigned roles, which determine how they can access the MTS data module's interface. The MTS data module implements the *IsCallerInRole* method, which you lets you check the role of the currently connected client and conditionally allow certain functions based on that role. For more information about MTS security, see Role-based security.
- **Database handle pooling.** MTS data modules automatically pool database connections so that when one client is finished with a database connection, another client can reuse it. This cuts down on network traffic, because your middle tier does not need to log off of the remote database server and then log on again. When pooling database handles, your database component should set the *KeepConnection* property to *False*, so that your application maximizes the sharing of connections.
- **MTS transactions**. When using MTS, you can integrate your own MTS transactions into the application server to provide enhanced transaction support. MTS transactions can span multiple databases, or include functions that do not involve databases at all. For more information about MTS transactions, see Managing transactions in multi-tiered applications.
- **Just-in-time activation and as-soon-as-possible deactivation.** You can write your MTS server so that remote data module instances are activated and deactivated on an as-needed basis. When using just-in-time activation and as-soon-as-possible deactivation, your remote data module is instantiated only when it is needed to handle client requests. This prevents it from tying up database handles when they are not in use.

  Using just-in-time activation and as-soon-as-possible deactivation provides a middle ground between routing all clients through a single remote data module instance, and creating a separate instance for every client connection. With a single remote data module instance, the application server must handle all database calls through a single database connection. This acts as a bottleneck, and can impact performance when there are many clients. With multiple instances of the remote data module, each instance can maintain a separate database connection, thereby avoiding the need to serialize database access. However, this monopolizes resources because other clients can't use the database connection while it is associated with another client's remote data module.

To take advantage of transactions, just-in-time activation, and as-soon-as-possible deactivation, remote data module instances must be stateless. This means you must provide additional support if your client relies on state information. For example, the client must pass information about the current record when performing incremental fetches. For more information about state information and remote data modules in multi-tiered applications, see Supporting state information in remote data modules.

By default, all automatically generated calls to an MTS data module are transactional (that is, they assume that when the call exits, the MTS data module can be deactivated and any current transactions can be committed or rolled back). You can write an MTS data module that depends on persistent state information by setting the *AutoComplete* property to *False*, but it will not support transactions, just-in-time activation, or as-soon-as-possible deactivation.

Warning: When using MTS, database connections should not be opened until the remote data module is activated. While developing your application, be sure that all datasets are not active and the database is not connected before running your application. In the application itself, you must add code to open database connections when the data module is activated and to close them when the data module is deactivated.

Pooling remote data modules

Topic groups See also

Object pooling allows you some of the benefits available from MTS when you are not using DCOM. Under object pooling, you can limit the number of instances of your remote data module that are created. This limits the number of database connections that you must hold, as well as any other resources used by the remote data module.

When the server receives client requests, it passes them on to the first available remote data module in the pool. If there is no available remote data module, it creates a new one (up to a maximum number that you specify). This provides a middle ground between routing all clients through a single remote data module instance (which can act as a bottleneck), and creating a separate instance for every client connection (which can consume many resources).

If a remote data module instance in the pool does not receive any client requests for a while, it is automatically freed. This prevents the pool from monopolizing resources unless they are used.

Because a single instance of a remote data module potentially handles requests from several clients, it must not rely on persistent state information. See Supporting state information in remote data modules for more information on how to ensure that your remote data module is stateless.

To take advantage of object pooling, your remote data module must override the *UpdateRegistry* method. In the overridden method, you can call *RegisterPooled* when the remote data module registers and *UnregisterPooled* when the remote data module unregisters.

You can only take advantage of object pooling when the connection is formed using HTTP.

Using the IAppServer interface

Topic groups See also

Remote data modules on the application server support the *IAppServer* interface. Connection components on client applications look for this interface to form connections.

IAppServer provides the bridge between client applications and the provider components in the application server. Most client applications do not use *IAppServer* directly, but invoke it indirectly through the properties and methods of the client dataset. However, when necessary, you can make direct calls to the *IAppServer* interface by using the *AppServer* property of the client dataset.

The following table lists the methods of the *IAppServer* interface, as well as the corresponding methods and events on the provider component and the client dataset. These *IAppServer* methods include a *Provider* parameter to indicate which provider on the application server should provide data or resolve updates. In addition, most methods include an OleVariant parameter called *OwnerData* that allows the client application and application server to pass custom information back and forth. *OwnerData* is not used by default, but is passed to all event handlers so that you can write code that allows your application server to adjust for this information before and after each client call.

IAppServer	Provider component	TClientDataSet
AS_ApplyUpdates method	ApplyUpdates method, BeforeApplyUpdates event, AfterApplyUpdates event	ApplyUpdates method, BeforeApplyUpdates event, AfterApplyUpdates event.
AS_DataRequest method	DataRequest method, OnDataRequest event	DataRequest method.
AS_Execute method	Execute method, BeforeExecute event, AfterExecute event	Execute method, BeforeExecute event, AfterExecute event.
AS_GetParams method	GetParams method, BeforeGetParams event, AfterGetParams event	FetchParams method, BeforeGetparams event, AfterGetParams event.
AS_GetProviderNames method	Used to identify all available providers.	Used to create a design-time list for ProviderName property.
AS_GetRecords method	GetRecords method, BeforeGetRecords event, AfterGetRecords event	GetNextPacket method, Data property, BeforeGetRecords event, AfterGetRecords event
AS_RowRequest method	RowRequest method, BeforeRowRequest event, AfterRowRequest event	FetchBlobs method, FetchDetails method, RefreshRecord method, BeforeRowRequest event, AfterRowRequest event

Choosing a connection protocol

Topic groups See also

Each communications protocol you can use to connect your client applications to the application server provides its own unique benefits. Before choosing a protocol, consider how many clients you expect, how you are deploying your application, and future development plans.

The following topics describe the unique features for each connection protocol:

- Using DCOM connections
- Using Socket connections
- Using Web connections
- Using OLEnterprise
- Using CORBA connections

Using DCOM connections

Topic groups See also

DCOM provides the most direct approach to communication, requiring no additional runtime applications on the server. However, because DCOM is not included with Windows 95, client machines may not have DCOM installed.

DCOM provides the only approach that lets you use MTS security. MTS security is based on assigning roles to the callers of MTS objects. When calling into MTS using DCOM, DCOM informs MTS about the client application that generated the call. MTS can then accurately determine the role of the caller. When using other protocols, however, there is a runtime executable, separate from the application server, that receives client calls. This runtime executable makes COM calls into the application server on behalf of the client. MTS can't assign roles to separate clients because, as far as MTS can tell, all calls to the application server are made by the runtime executable. For more information about MTS security, see Role-based security.

Using Socket connections

Topic groups See also

TCP/IP Sockets let you create lightweight clients. For example, if you are writing a Web-based client application, you can't be sure that client systems support DCOM. Sockets provide a lowest common denominator that you know will be available for connecting to the application server. For more information about Sockets, see Working with sockets.

Instead of instantiating the remote data module directly from the client (as happens with DCOM), sockets use a separate application on the server (ScktSrvr.exe), which accepts client requests and instantiates the remote data module using COM. The connection component on the client and ScktSrvr.exe on the server are responsible for marshaling *IAppServer* calls.

Note: ScktSrvr.exe can run as an NT service application. Register it with the Service manager by starting it using the -install command line option. You can unregister it using the -uninstall command line option.

Before you can use a socket connection, the application server must register its availability to clients using a socket connection. By default, all new remote data modules automatically register themselves by adding a call to *EnableSocketTransport* in the *UpdateRegistry* method. You can remove this call to prevent socket connections to your application server.

Note: Because older servers did not add this registration, you can disable the check for whether an application server is registered by unchecking the Connections|Registered Objects Only menu item on ScktSrvr.exe.

When using sockets, there is no protection on the server against client systems failing before they release a reference to interfaces on the application server. While this results in less message traffic than when using DCOM (which sends periodic keep-alive messages), this can result in an application server that can't release its resources because it is unaware that the client has gone away.

Using Web connections

Topic groups See also

HTTP lets you create clients that can communicate with an application server that is protected by a “firewall”. HTTP messages provide controlled access to internal applications so that you can distribute your client applications safely and widely. Like Socket connections, HTTP messages provide a lowest common denominator that you know will be available for connecting to the application server. For more information about HTTP messages, see Creating Internet server applications.

Instead of instantiating the remote data module directly from the client (as happens with DCOM), HTTP-based connections use a Web server application on the server (httpsrvr.dll) which accepts client requests and instantiates the remote data module using COM. Because of this, they are also called Web connections. The connection component on the client and httpsrvr.dll on the server are responsible for marshaling *IAppServer* calls.

Web connections can take advantage of the SSL security provided by wininet.dll (a library of internet utilities that runs on the client system). Once you have configured the Web server on the server system to require authentication, you can specify the user name and password using the properties of the Web connection component.

As an additional security measure, the application server must register its availability to clients using a Web connection. By default, all new remote data modules automatically register themselves by adding a call to *EnableWebTransport* in the *UpdateRegistry* method. You can remove this call to prevent Web connections to your application server.

Web connections can take advantage of object pooling. This allows your server to create a limited pool of remote data module instances that are available for client requests. By pooling the remote data modules, your server does not consume the resources for the data module and its database connection except when they are needed.

Unlike other connection components, you can’t use callbacks when the connection is formed via HTTP.

Using OLEnterprise

Topic groups See also

OLEnterprise lets you use the Business Object Broker instead of relying on client-side brokering. The Business Object Broker provides load-balancing, fail-over, and location transparency.

When using OLEnterprise, you must install OLEnterprise runtime on both client and server systems. OLEnterprise runtime handles the marshaling of Automation calls and communicates between the client and server system using remote procedure calls (RPCs). For more information, see the OLEnterprise documentation.

Using CORBA connections

Topic groups See also

CORBA lets you integrate your multi-tiered database applications into an environment that is standardized on CORBA. For example, the MIDAS client for Java components rely on a CORBA connection. Because CORBA (and Java) is available on multiple platforms, this allows you to write cross-platform MIDAS applications. For more information about using CORBA in Delphi, see Writing CORBA applications.

By using CORBA, your application automatically gets the benefits of load-balancing, location transparency, and fail-over from the ORB runtime software. In addition, you can add hooks to take advantage of other CORBA services.

Building a multi-tiered application

Topic groups See also

The general steps for creating a multi-tiered database application are

1. Create the application server.
2. Register or install the application server.

- If the application server uses DCOM, HTTP, sockets, or OLEnterprise as a communication protocol, it acts as an Automation server and must be registered like any other ActiveX or COM server. For information about registering an application, see Registering an application as an Automation server.
- If you are using MTS, the application server must be an Active Library rather than an .EXE. Because all COM calls must go through the MTS proxy, you do not register the application server. Instead, you install it with MTS. For information about installing libraries with MTS, see Installing MTS objects into an MTS package.
- When the application server uses CORBA, registration is optional. If you want to allow client applications to use dynamic binding to your interface, you must install the server's interface in the Interface Repository. In addition, if you want to allow client applications to launch the application server when it is not already running, it must be registered with the OAD (Object Activation Daemon). For more information about registering a CORBA server, see Registering server interfaces.

3. Create a client application.

The order of creation is important. You should create and run the application server before you create a client. At design time, you can then connect to the application server to test your client. You can, of course, create a client without specifying the application server at design time, and only supply the server name at runtime. However, doing so prevents you from seeing if your application works as expected when you code at design time, and you will not be able to choose servers and providers using the Object Inspector.

Note: If you are not creating the client application on the same system as the server, and you are not using a Web connection or socket connection, you may want to register or install the application server on the client system. This makes the connection component aware of the application server at design time so that you can choose server names and provider names from a drop-down list in the Object Inspector. (If you are using a Web connection or socket connection, the connection component fetches the names of registered servers from the server machine.)

The following topics describe how you can enhance the basic multi-tiered architecture to support the special needs of your application:

- Extending the application server's interface
- Managing transactions in multi-tiered applications
- Supporting master/detail relationships
- Supporting state information in remote data modules
- Writing MIDAS Web applications

Creating the application server

Topic groups See also

You create an application server very much as you create most database applications. The major difference is that the application server includes a dataset provider.

To create an application server, start a new project, save it, and follow these steps:

1. Add a new remote data module to the project. From the main menu, choose File|New. Choose the Multitier page in the new items dialog, and select
   - **Remote Data Module** if you are creating a COM Automation server that clients access using DCOM, HTTP, sockets, or OLEnterprise.
   - **MTS Data Module** if you are creating an Active Library that clients access using MTS. Connections can be formed using DCOM, HTTP, sockets, or OLEnterprise.
   - **CORBA Data Module** if you are creating a CORBA server.

   For more detailed information about setting up a remote data module, see Setting up the remote data module.

   Note: Remote data modules are more than simple data modules. The CORBA remote data module acts as a CORBA server. Other data modules are COM Automation objects.
2. Place the appropriate dataset components on the data module and set them up to access the database server.
3. Place a *TDataSetProvider* component on the data module for each dataset. This provider is required for brokering client requests and packaging data.
4. Set the *DataSet* property for each provider component to the name of the dataset to access. There are additional properties that you can set for the provider. For more detailed information about setting up a provider, see Using provider components.
5. Write application server code to implement events, shared business rules, shared data validation, and shared security. You may want to extend the application server's interface to provide additional ways that the client application can call the server.
6. Save, compile, and register or install the application server.
   - When the application server uses DCOM, HTTP, sockets, or OLEnterprise as a communication protocol, it acts as an Automation server and must be registered like any other ActiveX or COM server. For information about registering an application, see Registering an application as an Automation server.
   - If you are using MTS, the application server must be an Active Library rather than an .EXE. Because all COM calls must go through the MTS proxy, you do not register the application server. Instead, you install it with MTS. For information about installing libraries with MTS, see Installing MTS objects into an MTS package.
   - When the application server uses CORBA, registration is optional. If you want to allow client applications to use dynamic (late) binding to your interface, you must install the server's interface in the Interface Repository. In addition, if you want to allow client applications to launch the application server when it is not already running, it must be registered with the OAD (Object Activation Daemon). For more information about registering CORBA servers, see Registering server interfaces.
7. If your server application does not use DCOM, you must install the runtime software that receives client messages, instantiates the remote data module, and marshals interface calls.
   - For TCP/IP sockets this is a socket dispatcher application, Scktsrvr.exe.
   - For HTTP connections this is httpsrvr.dll, an ISAPI/NSAPI DLL that must be installed with your Web server.
   - For OLEnterprise, this is the OLEnterprise runtime.
   - For CORBA, this is the VisiBroker ORB.

Setting up the remote data module

Topic groups See also

When you set up and run an application server, it does not establish any connection with client applications. Instead, connection is maintained by client applications. The client application uses its connection component to establish a connection to the application server, which it uses to communicate with its selected provider. All of this happens automatically, without your having to write code to manage incoming requests or supply interfaces.

When you create the remote data module, you must provide certain information that indicates how it responds to client requests. This information varies, depending on the type of remote data module. See The structure of the application server for information on what type of remote data module you need.

The following topics describe how to configure each type of remote data module:

- Configuring TRemoteDataModule
- Configuring TMTSDataModule
- Configuring TCorbaDataModule

Configuring TRemoteDataModule

Topic groups See also

To add a *TRemoteDataModule* component to your application, choose File|New and select Remote Data Module from the Multitier page of the new items dialog. You will see the Remote Data Module wizard.

You must supply a class name for your remote data module. This is the base name of a descendant of *TRemoteDataModule* that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name *MyDataServer*, the wizard creates a new unit declaring *TMyDataServer*, a descendant of *TRemoteDataModule*, which implements *IMyDataServer*, a descendant of *IAppServer*.

Note: You can add your own properties and methods to the new interface. For more information, see Extending the application server's interface.

If you are creating a DLL (Active Library), you must specify the threading model in the Remote Data Module wizard. You can choose Single-threaded, Apartment-threaded, Free-threaded, or Both.

- If you choose Single-threaded, COM ensures that only one client request is serviced at a time. You do not need to worry about client requests interfering with each other.
- If you choose Apartment-threaded, COM ensures that any instance of your remote data module services one request at a time. When writing code in an Apartment-threaded library, you must guard against thread conflicts if you use global variables or objects not contained in the remote data module. This is the recommended model if you are using BDE-enabled datasets. (Note that you will need a session component with its *AutoSessionName* property set to *True* to handle threading issues on BDE-enabled datasets)
- If you choose Free-threaded, your application can receive simultaneous client requests on several threads. You are responsible for ensuring your application is thread-safe. Because multiple clients can access your remote data module simultaneously, you must guard your instance data (properties, contained objects, and so on) as well as global variables. This is the recommended model if you are using ADO datasets.
- If you choose Both, your library works the same as when you choose Free-threaded, with one exception: all callbacks (calls to client interfaces) are serialized for you.

If you are creating an EXE, you must specify what type of instancing to use. You can choose Single instance or Multiple instance (Internal instancing applies only if the client code is part of the same process space.)

- If you choose Single instance, each client connection launches its own instance of the executable. That process instantiates a single instance of the remote data module, which is dedicated to the client connection.
- If you choose Multiple instance, a single instance of the application (process) instantiates all remote data modules created for clients. Each remote data module is dedicated to a single client connection, but they all share the same process space.

Configuring TMTSDataModule

Topic groups See also

To add a *TMTSDataModule* component to your application, choose File|New and select MTS Data Module from the Multitier page of the new items dialog. You will see the MTS Data Module wizard.

You must supply a class name for your remote data module. This is the base name of a descendant of *TMTSDataModule* that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name *MyDataServer*, the wizard creates a new unit declaring *TMyDataServer*, a descendant of *TMTSDataModule*, which implements *IMyDataServer*, a descendant of *IAppServer*.

Note: You can add your own properties and methods to your new interface. For more information, see Extending the application server's interface.

MTS applications are always DLLs (Active Libraries). You must specify the threading model in the MTS Data Module wizard. Choose Single, Apartment, Free, or Both.

- If you choose Single, MTS ensures that only one client request is serviced at a time. You do not need to worry about client requests interfering with each other.
- If you choose Apartment or Free, you get the same thing: MTS ensures that any instance of your remote data module services one request at a time, but calls do not always use the same thread. You can't use thread variables, because there is no guarantee that subsequent calls to the remote data module instance will use the same thread. You must guard against thread conflicts if you use global variables or objects not contained in the remote data module. Instead of using global variables, you can use the shared property manager. For more information on the shared property manager, see Resource dispensers.
- If you choose Both, MTS calls into the remote data module's interface in the same way as when you choose Apartment or Free. However, any callbacks you make to client applications are serialized, so that you don't need to worry about them interfering with each other.

Note: The Apartment and Free models under MTS are different than the corresponding models under DCOM.

You must also specify the MTS transaction attributes of your remote data module. You can choose from the following options:

- Requires a transaction. When you select this option, every time a client uses your remote data module's interface, that call is executed in the context of an MTS transaction. If the caller supplies a transaction, a new transaction need not be created.
- Requires a new transaction. When you select this option, every time a client uses your remote data module's interface, a new transaction is automatically created for that call.
- Supports transactions. When you select this option, your remote data module can be used in the context of an MTS transaction, but the caller must supply the transaction when it invokes the interface.
- Does not support transactions. When you select this option, your remote data module can't be used in the context of MTS transactions.

Configuring TCORBADataModule

Topic groups See also

To add a *TCorbaDataModule* component to your application, choose File|New and select CORBA Data Module from the Multitier page of the new items dialog. You will see the CORBA Data Module wizard.

You must supply a class name for your remote data module. This is the base name of a descendant of *TCorbaDataModule* that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name *MyDataServer*, the wizard creates a new unit declaring *TMyDataServer*, a descendant of *TCorbaDataModule*, which implements *IMyDataServer*, a descendant of *IAppServer*.

Note: You can add your own properties and methods to your new interface. For more information on adding to your data module's interface, see Extending the application server's interface.

The CORBA Data Module wizard lets you specify how you want your server application to create instances of the remote data module. You can choose either shared or instance-per-client.

- When you choose shared, your application creates a single instance of the remote data module that handles all client requests. This is the model used in traditional CORBA development.
- When you choose instance-per-client, a new remote data module instance is created for each client connection. This instance persists until its timeout period elapses with no messages from the client. This allows the server to free instances when they are no longer used by clients, but holds the risk that the server may be freed prematurely if the client does not use the server's interface for a long time.

Note: Unlike instancing for COM servers, where the model determines the number of instances of the process that run, with CORBA, instancing determines the number of instances created of your object. They are all created within a single instance of the server executable.

In addition to the instancing model, you must specify the threading model in the CORBA Data Module wizard. You can choose Single- or Multi-threaded.

- If you choose Single-threaded, each remote data module instance is guaranteed to receive only one client request at a time. You can safely access the objects contained in your remote data module. However, you must guard against thread conflicts when you use global variables or objects not contained in the remote data module.
- If you choose Multi-threaded, each client connection has its own dedicated thread. However, your application may be called by multiple clients simultaneously, each on a separate thread. You must guard against simultaneous access of instance data as well as global memory. Writing Multi-threaded servers is tricky when you are using a shared remote data module instance, because you must protect all use of objects contained in your remote data module.

Creating a data provider for the application server

Topic groups See also

Each remote data module on an application server contains one or more provider components. Each client dataset uses a specific provider, which acts as the bridge between the client dataset and the data it represents. A provider component (*TDataSetProvider*) takes care of packaging data into data packets that it sends to clients and applying updates received from the client.

Most of the data logic in the application server is handled by the provider components contained in the remote data module. Event handlers that respond to client requests implement your business and data logic, while properties on the provider component control what information is included in data packets. See Creating multi-tiered applications for details on how to use a provider component to control the interaction with client applications.

Extending the interface of the application server

Topic groups See also

Client applications interact with the application server by creating or connecting to an instance of the remote data module. They use its interface as the basis of all communication with the application server.

You can add to your remote data module's interface to provide additional support for your client applications. This interface is a descendant of *IAppServer* and is created for you automatically by the wizard when you create the remote data module.

To add to the remote data module's interface, you can

- Choose the Add to Interface command from the Edit menu in the IDE. Indicate whether you are adding a procedure, function, or property, and enter its syntax. When you click OK, you will be positioned in the code editor on the implementation of your new interface member.
- Use the type library editor. Select the interface for your application server in the type library editor, and click the tool button for the type of interface member (method or property) that you are adding. Give your interface member a name in the Attributes page, specify parameters and type in the Parameters page, and then refresh the type library. For more information about using the type library editor, see Working with type libraries. Note that many of the features you can specify in the type library editor (such as help context, version, and so on) do not apply to CORBA interfaces. Any values you specify for these in the type library editor are ignored.

What Delphi does when you add new entries to the interface depends on whether you are creating a COM-based (*TRemoteDataModule* or *TMTSDataModule*) or CORBA (*TCorbaDataModule*) server.

- When you add to a COM interface, your changes are added to your unit source code and the type library file (.TLB).
- When you add to a CORBA interface, your changes are reflected in your unit source code and the automatically generated _TLB unit. The _TLB unit is added to the **uses** clause of your unit. You must add this unit to the **uses** clause in your client application if you want to take advantage of early binding. In addition, you can save an .IDL file from the type library editor using the Export to IDL button. The .IDL file is needed for registering the interface with the Interface Repository and Object Activation Daemon.

Note: You must explicitly save the TLB file by choosing Refresh in the type library editor and then saving the changes from the IDE.

Once you have added to your remote data module's interface, locate the properties and methods that were added to your remote data module's implementation. Add code to finish this implementation.

Client applications call your interface extensions using the *AppServer* property of their connection component. For more information on how to do this, see Calling server interfaces.

Adding callbacks to the application server's interface

You can allow the application server to call your client application by introducing a callback. To do this, the client application passes an interface to one of the application server's methods, and the application server later calls this method as needed. However, if your extensions to the remote data module's interface include callbacks, you can't use an HTTP-based connection. *TWebConnection* does not support callbacks. If you are using a socket-based connection, client applications must indicate whether they are using callbacks by setting the *SupportCallbacks* property. All other types of connection automatically support callbacks.

Extending the application server's interface when using MTS

When using transactions or just-in-time activation under MTS, you must be sure all new methods call *SetComplete* to tell MTS when it is finished. This allows transactions to complete and so permits the remote data module to be deactivated. Furthermore, you can't return any values from your new methods that allow the client to communicate directly with objects or interfaces on the application server. This is because any communication that does not go through the remote data module's interface bypasses the MTS proxy, which can invalidate transactions. If you are using a stateless MTS data module, bypassing the MTS proxy can lead to crashes because you can't guarantee that the remote data module is active.

Creating the client application

Topic groups See also

In most regards, creating a multi-tiered client application is similar to creating a traditional two-tiered client. The major differences are that a multi-tiered client uses

- A connection component to establish a conduit to the application server.
- One or more *TClientDataSet* components to link to a data provider on the application server.

Data-aware controls on the client are connected through data source components to these client datasets instead of *TTable*, *TQuery*, *TStoredProc* or *TADODataSet* components.

To create a multi-tiered client application, start a new project and follow these steps:

1. Add a new data module to the project.
2. Place a connection component on the data module. The type of connection component you add depends on the communication protocol you want to use. See The structure of the client application for details.
3. Set properties on your connection component to specify the application server with which it should establish a connection. To learn more about setting up the connection component, see Connecting to the application server.
4. Set the other connection component properties as needed for your application. For example, you might set the *ObjectBroker* property to allow the connection component to choose dynamically from several servers. For more information about using the connection components, see Managing server connections.
5. Place as many *TClientDataSet* components as needed on the data module, and set the *RemoteServer* property for each component to the name of the connection component you placed in Step 2. For a full introduction to client datasets, see Creating and using a client dataset.
6. Set the *ProviderName* property for each *TClientDataSet* component. If your connection component is connected to the application server at design time, you can choose available application server providers from the *ProviderName* property's drop-down list.
7. Create the client application in much the same way you would create any other database application. You will probably want to use some of the special features of client datasets that support their interaction with the provider components on the application server. These are described in Using a client dataset with a data provider.

Connecting to the application server

Topic groups See also

To establish and maintain a connection to an application server, a client application uses one or more connection components. You can find these components on the MIDAS page of the Component palette.

Use a connection component to

- Identify the protocol for communicating with the application server. Each type of connection component represents a different communication protocol. See Choosing a connection protocol for details on the benefits and limitations of the available protocols.
- Indicate how to locate the server machine. The details of identifying the server machine vary depending on the protocol. See the following topics for details:
  - Specifying a connection using DCOM
  - Specifying a connection using sockets
  - Specifying a connection using HTTP
  - Specifying a connection using OLEnterprise
  - Specifying a connection using CORBA
- Identify the application server on the server machine.

  If you are not using CORBA, identify the server using the *ServerName* or *ServerGUID* property. *ServerName* identifies the base name of the class you specify when creating the remote data module on the application server. See Setting up the remote data module for details on how this value is specified on the server. If the server is registered or installed on the client machine, or if the connection component is connected to the server machine, you can set the *ServerName* property at design time by choosing from a drop-down list in the Object Inspector. *ServerGUID* specifies the GUID of the remote data module's interface. You can look up this value using the type library editor.

  If you are using CORBA, identify the server using the *RepositoryID* property. *RepositoryID* specifies the Repository ID of the application server's factory interface, which appears as the third argument in the call to *TCorbaVCLComponentFactory.Create* that is automatically added to the initialization section of the CORBA server's implementation unit. You can also set this property to the base name of the CORBA data module's interface (the same string as the *ServerName* property for other connection components), and it is automatically converted into the appropriate Repository ID for you.
- Manage server connections. Connection components can be used to create or drop connections and to call application server interfaces.

Usually the application server is on a different machine from the client application, but even if the server resides on the same machine as the client application (for example, during the building and testing of the entire multi-tier application), you can still use the connection component to identify the application server by name, specify a server machine, and use the application server interface.

Specifying a connection using DCOM

Topic groups See also

When using DCOM to communicate with the application server, client applications include a *TDCOMConnection* component for connecting to the application server. *TDCOMConnection* uses the *ComputerName* property to identify the machine on which the server resides.

When *ComputerName* is blank, the DCOM connection component assumes that the application server resides on the client machine or that the application server has a system registry entry. If you do not provide a system registry entry for the application server on the client when using DCOM, and the server resides on a different machine from the client, you must supply *ComputerName*.

Note: Even when there is a system registry entry for the application server, you can specify *ComputerName* to override this entry. This can be especially useful during development, testing, and debugging.

If you have multiple servers that your client application can choose from, you can use the *ObjectBroker* property instead of specifying a value for *ComputerName*. For more information, see Brokering connections.

If you supply the name of a host computer or server that cannot be found, the DCOM connection component raises an exception when you try to open the connection.

Specifying a connection using sockets

Topic groups See also

You can establish a connection to the application server using sockets from any machine that has a TCP/IP address. This method has the advantage of being applicable to more machines, but does not provide for using any security protocols. When using sockets, include a *TSocketConnection* component for connecting to the application server.

TSocketConnection identifies the server machine using the IP Address or host name of the server system, and the port number of the socket dispatcher program (Scktsrvr.exe) that is running on the server machine. For more information about IP addresses and port values, see Describing sockets.

Three properties of *TSocketConnection* specify this information:

- *Address* specifies the IP Address of the server.
- *Host* specifies the host name of the server.
- *Port* specifies the port number of the socket dispatcher program on the application server.

Address and *Host* are mutually exclusive. Setting one unsets the value of the other. For information on which one to use, see Describing the host.

If you have multiple servers that your client application can choose from, you can use the *ObjectBroker* property instead of specifying a value for *Address* or*Host*. For more information, see Brokering connections.

By default, the value of *Port* is 211, which is the default port number of the socket dispatcher programs supplied with Delphi. If the socket dispatcher has been configured to use a different port, set the *Port* property to match that value.

Note: You can configure the port of the socket dispatcher while it is running by right-clicking the Borland Socket Server tray icon and choosing Properties.

Although socket connections do not provide for using security protocols, you can customize the socket connection to add your own encryption. To do this, create and register a COM object that supports the *IDataIntercept* interface. This is an interface for encrypting and decrypting data. Next, set the *InterceptGUID* property of the socket connection component to the GUID for this COM object. Finally, right click the Borland Socket Server tray icon, choose Properties, and on the properties tab set the Intercept GUID to the same GUID. This mechanism can also be used for data compression and decompression.

Specifying a connection using HTTP

Topic groups See also

You can establish a connection to the application server using HTTP from any machine that has a TCP/IP address. Unlike sockets, however, HTTP allows you to take advantage of SSL security and to communicate with a server that is protected behind a firewall. When using HTTP, include a *TWebConnection* component for connecting to the application server.

The Web connection component establishes a connection to the Web server application (httpsrvr.dll), which in turn communicates with the application server. *TWebConnection* locates httpsrvr.dll using a Uniform Resource Locator (URL). The URL specifies the protocol (http or, if you are using SSL security, https), the host name for the machine that runs the Web server and httpsrvr.dll, and the path to the Web server application (Httpsrvr.dll). Specify this value using the URL property.

Note: When using *TWebConnection*, wininet.dll must be installed on the client machine. If you have IE3 or higher installed, wininet.dll can be found in the Windows system directory.

If the Web server requires authentication, or if you are using a proxy server that requires authentication, you must set the values of the *UserName* and *Password* properties so that the connection component can log on.

If you have multiple servers that your client application can choose from, you can use the *ObjectBroker* property instead of specifying a value for URL. For more information, see Brokering connections.

Specifying a connection using OLEnterprise

Topic groups See also

When using OLEnterprise to communicate with the application server, client applications should include a *TOLEnterpriseConnection* component for connecting to the application server. When using OLEnterprise, you can either connect directly to the server machine, or you can use the Business Object Broker.

- To use OLEnterprise without going through a Business Object Broker, set the *ComputerName* property to the name of the server machine, just as you would use the *ComputerName* property for a DCOM connection.
- To use the load-balancing and fail-over services of the Business Object Broker, set the *BrokerName* property to the name of the Business Object Broker.

ComputerName and *BrokerName* are mutually exclusive. Setting the value of one unsets the value of the other.

For more information about using OLEnterprise, see the OLEnterprise documentation.

Specifying a connection using CORBA

Topic groups See also

Only the *RepositoryID* property is necessary in order to specify a CORBA connection. This is because a Smart Agent on the local network automatically locates an available server for your CORBA client.

However, you can limit the possible servers to which your client application connects by the other properties of the CORBA connection component. If you want to specify a particular server machine, rather than letting the CORBA Smart Agent locate any available server, use the *HostName* property. If there is more than one object instance that implements your server interface, you can specify which object you want to use by setting the *ObjectName* property.

The *TCorbaConnection* component obtains an interface to the CORBA data module on the application server in one of two ways:

- If you are using early (static) binding, you must add the _TLB.pas file (generated by the type library editor) to your client application. Early binding is highly recommended, both for compile-time type checking and because it is much faster than late (dynamic) binding.
- If you are using late (dynamic) binding, the interface must be registered with the Interface Repository. For more information about registering an interface with the Interface Repository, see Writing CORBA clients.

For more information on early vs. late binding, see Calling server interfaces.

Brokering connections

Topic groups See also

If you have multiple servers that your client application can choose from, you can use an Object Broker to locate an available server system. The object broker maintains a list of servers from which the connection component can choose. When the connection component needs to connect to an application server, it asks the Object Broker for a computer name (or IP address, host name, or URL). The broker supplies a name, and the connection component forms a connection. If the supplied name does not work (for example, if the server is down), the broker supplies another name, and so on, until a connection is formed.

Once the connection component has formed a connection with a name supplied by the broker, it saves that name as the value of the appropriate property (*ComputerName*, *Address*, *Host, RemoteHost*, or URL). If the connection component closes the connection later, and then needs to reopen the connection, it tries using this property value, and only requests a new name from the broker if the connection fails.

Use an Object Broker by specifying the *ObjectBroker* property of your connection component. When the *ObjectBroker* property is set, the connection component does not save the value of *ComputerName*, *Address*, *Host*, *RemoteHost*, or URL to disk.

Note: Do not use the *ObjectBroker* property with OLEnterprise connections or CORBA connections. Both of these protocols have their own brokering services.

Managing server connections

Topic groups See also

The main purpose of connection components is to locate and connect to the application server. Because they manage server connections, you can also use connection components to call the methods of the application server's interface.

The following topics describe how to use a connection component for

- Connecting to the server.
- Dropping or changing a server connection.
- Calling server interfaces.

Connecting to the server

Topic groups See also

To locate and connect to the application server, you must first set the properties of the connection component to identify the application server. This process is described in Connecting to the application server. In addition, before opening the connection, any client datasets that use the connection component to communicate with the application server should indicate this by setting their *RemoteServer* property to specify the connection component.

The connection is opened automatically when client datasets try to access the application server. For example, setting the *Active* property of the client dataset to *True* opens the connection, as long as the *RemoteServer* property has been set.

If you do not link any client datasets to the connection component, you can open the connection by setting the *Connected* property of the connection component to *True*.

Before a connection component establishes a connection to an application server, it generates a *BeforeConnect* event. You can perform any special actions prior to connecting in a *BeforeConnect* handler that you code. After establishing a connection, the connection component generates an *AfterConnect* event for any special actions.

Dropping or changing a server connection

Topic groups See also

A connection component drops a connection to the application server when you

- set the *Connected* property to *False*.
- free the connection component. A connection object is automatically freed when a user closes the client application.
- change any of the properties that identify the application server (*ServerName*, *ServerGUID*, *ComputerName*, and so on). Changing these properties allows you to switch among available application servers at runtime. The connection component drops the current connection and establishes a new one.

Note: Instead of using a single connection component to switch among available application servers, a client application can instead have more than one connection component, each of which is connected to a different application server.

Before a connection component drops a connection, it automatically calls its *BeforeDisconnect* event handler, if one is provided. To perform any special actions prior to disconnecting, write a *BeforeDisconnect* handler. Similarly, after dropping the connection, the *AfterDisconnect* event handler is called. If you want to perform any special actions after disconnecting, write an *AfterDisconnect* handler.

Calling server interfaces

Topic groups See also

Applications do not need to call the *IAppServer* interface directly because the appropriate calls are made automatically when you use the properties and methods of the client dataset. However, while it is not necessary to work directly with the *IAppServer* interface, you may have added your own extensions to the remote data module's interface. When you extend the application server's interface, you need a way to call those extensions using the connection created by your connection component. You can do this using the *AppServer* property of the connection component. *AppServer* is a Variant that represents the application server's interface. You can call an interface method using *AppServer* by writing a statement such as

```
MyConnection.AppServer.SpecialMethod(x,y);
```

However, this technique provides late (dynamic) binding of the interface call. That is, the *SpecialMethod* procedure call is not bound until runtime when the call is executed. Late binding is very flexible, but by using it you lose many benefits such as code insight and type checking. In addition, late binding is slower than early binding, because the compiler generates additional calls to the server to set up interface calls before they are invoked.

When you are using DCOM or CORBA as a communications protocol, you can use early binding of *AppServer* calls. Use the **as** operator to cast the *AppServer* variable to the *IAppServer* descendant you created when you created the remote data module. For example:

```
with MyConnection.AppServer as IMyAppServer do
  SpecialMethod(x,y);
```

To use early binding under DCOM, the server's type library must be registered on the client machine. You can use TRegsvr.exe, which ships with Delphi to register the type library.

Note: See the TRegSvr demo (which provides the source for TRegsvr.exe) for an example of how to register the type library programmatically.

To use early binding with CORBA, you must add the _TLB unit that is generated by the type library editor to your project. To do this, add this unit to the **uses** clause of your unit.

When you are using TCP/IP or OLEnterprise, you can't use true early binding, but because the remote data module uses a dual interface, you can use the application server's dispinterface to improve performance over simple late-binding. The dispinterface has the same name as the remote data module's interface, with the string 'Disp' appended. You can assign the *AppServer* property to a variable of this type to obtain the dispinterface. Thus:

```
var
  TempInterface: IMyAppServerDisp;
begin
  TempInterface := MyConnection.AppServer;
...
  TempInterface.SpecialMethod(x,y);
...
end;
```

Note: To use the dispinterface, you must add the _TLB unit that is generated when you save the type library to the **uses** clause of your client module.

Managing transactions in multi-tiered applications

Topic groups See also

When client applications apply updates to the application server, the provider component automatically wraps the process of applying updates and resolving errors in a transaction. This transaction is committed if the number of problem records does not exceed the *MaxErrors* value specified as an argument to the *ApplyUpdates* method. Otherwise, it is rolled back.

In addition, you can add transaction support to your server application by adding a database component or using passthrough SQL. This works the same way that you would manage transactions in a two-tiered application. For more information about this sort of transaction control, see Using transactions and Working with (connection) transactions

If you are using MTS, you can broaden your transaction support by using MTS transactions. MTS transactions can include any of the business logic on your application server, not just the database access. In addition, because they support two-phase commits, MTS transactions can span multiple databases.

Warning: Two-phase commit is fully implemented only on Oracle7 and MS-SQL databases. If your transaction involves multiple databases, and some of them are remote servers other than Oracle7 or MS-SQL, your transaction runs a small risk of only partially succeeding. Within any one database, however, you will always have transaction support.

To use MTS transactions, extend the application server's interface to include method calls that encapsulate the transaction, if necessary. When configuring the MTS remote data module, indicate that it must participate in transactions. When a client calls a method on your application server's interface, it is automatically wrapped in a transaction. All client calls to your application server are then enlisted in that transaction until you indicate that the transaction is complete. These calls either succeed as a whole or are rolled back.

Note: Do not combine MTS transactions with explicit transactions created by a database component or using passthrough SQL. When your remote data module is enlisted in an MTS transaction, it automatically enlists all of your database calls in the transaction as well.

For more information about using MTS transactions, see MTS transaction support.

Supporting master/detail relationships

Topic groups See also

You can create master/detail relationships between client datasets in your client application in the same way you set up master/detail forms in one- and two-tiered applications. For more information about setting up master/detail forms, see Creating master/detail forms.

However, this approach has two major drawbacks:

- The detail table must fetch and store all of its records from the application server even though it only uses one detail set at a time. This problem can be mitigated by using parameters. For more information, see Limiting records with parameters.
- It is very difficult to apply updates, because client datasets apply updates at the dataset level and master/detail updates span multiple datasets. Even in a two-tiered environment, where you can use the database to apply updates for multiple tables in a single transaction, applying updates in master/detail forms is tricky. See Applying updates for master/detail tables for more information on applying updates in traditional master/detail forms.

In multi-tiered applications, you can avoid these problems by using nested tables to represent the master/detail relationship. To do this, set up a master/detail relationship between the tables on the application server. Then set the *DataSet* property of your provider component to the master table.

When clients call the *GetRecords* method of the provider, it automatically includes the detail datasets as a DataSet field in the records of the data packet. When clients call the *ApplyUpdates* method of the provider, it automatically handles applying updates in the proper order.

See Representing master/detail relationships for more information on using nested datasets to support master/detail relationships in client datasets.

Supporting state information in remote data modules

Topic groups See also

The *IAppServer* interface, which controls all communication between client datasets and providers on the application server, is mostly stateless. When an application is stateless, it does not “remember” anything that happened in previous calls by the client. This stateless quality is useful if you are pooling database connections under MTS, because your application server does not need to distinguish between database connections for persistent information such as record currency. Similarly, this stateless quality is important when you are sharing remote data module instances between many clients, as occurs with MTS just-in-time activation, object pooling, or typical CORBA servers.

However, there are times when you want to maintain state information between calls to the application server. For example, when requesting data using incremental fetching, the provider on the application server must “remember” information from previous calls (the current record).

This is not a problem if the remote data module is configured so that each client has its own instance. When each client has its own instance of the remote data module, there are no other clients to change the state of the data module between client calls.

However, it is reasonable to want the benefits of sharing remote data module instances while still managing persistent state information. For example, you may need to use incremental fetching to display a dataset that is too large to fit in memory at one time.

Before and after any calls to the *IAppServer* interface that the client dataset sends to the application server (AS_*ApplyUpdates*, AS_*Execute*, AS_*GetParams*, AS_*GetRecords*, or AS_*RowRequest*), it receives an event where it can send or retrieve custom state information. Similarly, before and after providers respond to these client-generated calls, they receive events where they can retrieve or send custom state information. Using this mechanism, you can communicate persistent state information between client applications and the application server, even if the application server is stateless. For example, to enable incremental fetching in a stateless application server, you can do the following:

- Use the client dataset’s *BeforeGetRecords* event to send the key value of the last record to the application server:

```
TDataModule1.ClientDataSet1BeforeGetRecords(Sender: TObject; var OwnerData:
OleVariant);
var
  CurRecord: TBookMark;
begin
  with Sender as TClientDataSet do
  begin
    CurRecord := GetBookmark; { save the current record }
    try
      Last; {locate the last record in the new packet }
      OwnerData := FieldValues['Key']; { Send key value to the application server }
      GotoBookmark(CurRecord); { return to current record }
    finally
      FreeBookmark(CurRecord);
    end;
  end;
end;
```

- On the server, use the provider’s *BeforeGetRecords* event to locate the appropriate set of records:

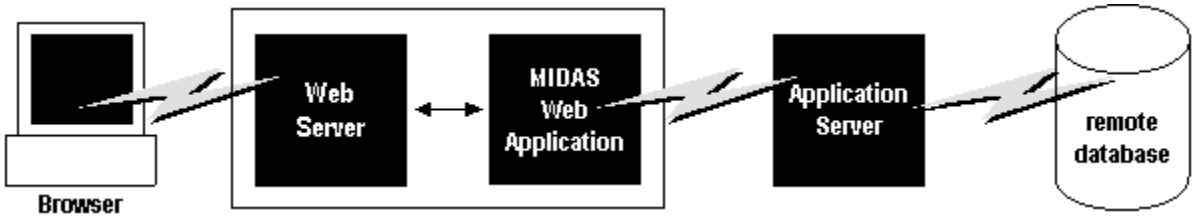
```
TRemoteDataModule1.Provider1BeforeGetRecords(Sender: TObject; var OwnerData:
OleVariant);
begin
  with Sender as TProvider do
    DataSet.Locate('Key', OwnerData, []);
end;
```

Note: The previous example uses a key value to mark the end of the record set rather than a bookmark. This is because bookmarks may not be valid between *IAppServer* calls if the server application pools database handles.

Writing MIDAS Web applications

Topic groups See also

If you want to create Web-based clients for your multi-tiered database application, you must replace the client tier with a special Web applications that acts simultaneously as a client to the application server and as a Web server application that is installed with a Web server on the same machine. This architecture is illustrated in the following figure.



There are two approaches that you can take to build the MIDAS Web application:

- You can combine the MIDAS architecture with Delphi's ActiveX support to distribute a client application as an ActiveX control. This allows any browser that supports ActiveX to run your client application as an in-process server.
- You can use XML data packets to build an InternetExpress application. This allows browsers that supports javascript to interact with your client application through html pages.

These two approaches are very different. Which one you choose depends on the following considerations:

- Each approach relies on a different technology (ActiveX vs. javascript and XML). Consider what systems your end-users will use. The first approach requires a browser to support ActiveX (which limits clients to a Windows platform). The second approach requires a browser to support javascript and the DHTML capabilities introduced by Netscape 4 and Internet Explorer 4.
- ActiveX controls must be downloaded to the browser to act as an in-process server. As a result, the clients using an ActiveX approach require much more memory than the clients of an html-based application.
- The InternetExpress approach can be integrated with other HTML pages. An ActiveX client must run in a separate window.
- The InternetExpress approach uses standard HTTP, thereby avoiding any firewall issues that confront an ActiveX application.
- The ActiveX approach provides greater flexibility in how you program your application. You are not limited by the capabilities of the javascript libraries. The client datasets used in the ActiveX approach surface more features (such as filters, ranges, aggregation, optional parameters, delayed fetching of BLOBs or nested details, and so on) than the XML brokers used in the InternetExpress approach.

Caution: Your Web client application may look and act differently when viewed from different browsers. Test your application with the browsers you expect your end-users to use.

Distributing a client application as an ActiveX control

Topic groups See also

The MIDAS architecture can be combined with Delphi's ActiveX features to distribute a client application as an ActiveX control.

When you distribute your client application as an ActiveX control, create the application server as you would for any other multi-tiered application. The only limitation is that you will want to use DCOM, HTTP, or sockets as a communications protocol, because you can't count on client machines having installed the OLEnterprise or CORBA runtime software.

When creating the client application, you must use an Active Form as the basis instead of an ordinary form. See Creating an Active Form for the client application for details.

Once you have built and deployed your client application, it can be accessed from any ActiveX-enabled Web browser on another machine. For a Web browser to successfully launch your client application, the Web server must be running on the machine that has the client application.

If the client application uses DCOM to communicate between the client application and the application server, the machine with the Web browser must be enabled to work with DCOM. If the machine with the Web browser is a Windows 95 machine, it must have installed DCOM95, which is available from Microsoft.

Creating an Active Form for the client application

Topic groups See also

1. Because the client application will be deployed as an ActiveX control, you must have a Web server that runs on the same system as the client application. You can use a ready-made server such as Microsoft's Personal Web server or you can write your own using the socket components described in "Working with sockets."
2. Create the client application following the steps described in "Creating the client application." except start by choosing File|New|Active Form, rather than beginning the client project as an ordinary Delphi project.
3. If your client application uses a data module, add a call to explicitly create the data module in the active form initialization.
4. When your client application is finished, compile the project, and select Project | Web Deployment Options. In the Web Deployment Options dialog, you must do the following:
    1. On the Project page, specify the Target directory, the URL for the target directory, and the HTML directory. Typically, the Target directory and the HTML directory will be the same as the projects directory for your Web Server. The target URL is typically the name of the server machine that is specified in the Windows Network|DNS settings.
    2. On the Additional Files page, include midas.dll with your client application.
5. Finally, select Project|WebDeploy to deploy the client application as an active form.

Any Web browser that can run Active forms can run your client application by specifying the .HTM file that was created when you deployed the client application. This .HTM file has the same name as your client application project, and appears in the directory specified as the Target directory.

Building Web applications using InternetExpress

Topic groups See also

MIDAS clients can request that the application server provide data packets that are coded in XML instead of OleVariants. By combining XML-coded data packets, special javascript libraries of database functions, and Delphi's Web server application support, you can create thin client applications that can be accessed using a Web browser that supports javascript. These applications make up Delphi's InternetExpress support.

Before building an InternetExpress application, you should understand Delphi's Web server application architecture and the MIDAS database architecture. These are described in Creating Internet server applications and Understanding MIDAS technology.

On the InternetExpress page of the component palette, you can find a set of components that extend this Web server application architecture to act as a MIDAS client. Using these components, the Web application generates HTML pages that contain a mixture of HTML, XML, and javascript. The HTML governs the layout and appearance of the pages seen by end users in their browsers. The XML encodes the data packets and delta packets that represent database information. The javascript allows the HTML controls to interpret and manipulate the data in these XML data packets.

If the InternetExpress application uses DCOM to connect to the application server, you must take additional steps to ensure that the application server grants access and launch permissions to its clients.

Tip: You can use the components on the InternetExpress page to build Web server applications with "live" data even if you do not have an application server. Simply add the provider and its dataset to the Web module.

Building an InternetExpress application

Topic groups See also

The following steps describe how to build a Web application that creates HTML pages for allowing users to interact with the data from an application server via a javascript-enabled Web browser.

1. Choose File|New to display the New Items dialog box, and on the New page select Web Server application. This process is described in Creating Web server applications.
2. From the MIDAS page of the component palette, add a connection component to the Web Module that appears when you create a new Web server application. The type of connection component you add depends on the communication protocol you want to use. See Choosing a connection protocol for details.
3. Set properties on your connection component to specify the application server with which it should establish a connection. To learn more about setting up the connection component, see Connecting to the application server.
4. Instead of a client dataset, add an XML broker from the InternetExpress page of the component palette to the Web module. Like *TClientDataSet*, *TXMLBroker* represents the data from a provider on the application server and interacts with the application server through its *IAppServer* interface. However, unlike client datasets, XML brokers request data packets as XML instead of as OleVariants and interact with InternetExpress components instead of data controls.
5. Set the *RemoteServer* property of the XML broker to point to the connection component you added in step 2. Set the *ProviderName* property to indicate the provider on the application server that provides data and applies updates. For more information about setting up the XML broker, see Using an XML broker.
6. Add a MIDAS page producer to the Web module for each separate page that users will see in their browsers. For each MIDAS page producer, you must set the *IncludePathURL* property to indicate where it can find the javascript libraries that augment its generated HTML controls with data management capabilities.
7. Right-click a Web page and choose Action Editor to display the Action editor. Add action items for every message you want to handle from browsers. Associate the page producers you added in step 6 with these actions by setting their *Producer* property or writing code in an *OnAction* event handler. For more information on adding action items using the Action editor, see Adding actions to the dispatcher.
8. Double-click each Web page to display the Web Page editor. (You can also display this editor by clicking the ellipses button in the Object Inspector next to the *WebPageItems* property.) In this editor you can add Web Items to design the pages that users see in their browsers. For more information about designing Web pages for your InternetExpress application, see Creating Web pages with a MIDAS page producer.
9. Build your Web application. Once you install this application with your Web server, browsers can call it by specifying the name of the application as the scriptname portion of the URL and the name of the Web Page component as the pathinfo portion.

Using the javascript libraries

Topic groups See also

The HTML pages generated by the InternetExpress components and the Web items they contain make use of several javascript libraries that ship with Delphi:

Library	Description
xmldom.js	This library is a DOM-compatible XML parser written in javascript. It allows parsers that do not support XML to use XML data packets. Note that this does not include support for XML Islands, which are supported by IE5 and later.
xmldb.js	This library defines data access classes that manage XML data packets and XML delta packets.
xmldisp.js	This library defines classes that associate the data access classes in xmldb with HTML controls in the HTML page.
xmlerrdisp.js	This library defines classes that can be used when reconciling update errors. These classes are not used by any of the built-in InternetExpress components, but are useful when writing a Reconcile producer.
xmlshow.js	This library includes functions to display formatted XML data packets and XML delta packets. This library is not used by any of the InternetExpress components, but is useful when debugging.

These libraries can be found in the Source/Webmidas directory. Once you have installed these libraries, you must set the *IncludePathURL* property of all MIDAS page producers to indicate where they can be found.

It is possible to write your own HTML pages using the javascript classes provided in these libraries instead of using Web items to generate your Web pages. However, you must ensure that your code does not do anything illegal, as these classes include minimal error checking (so as to minimize the size of the generated Web pages).

The classes in the javascript libraries are an evolving standard, and are updated regularly. If you want to use them directly rather than relying on Web items to generate the javascript code, you can get the latest versions and documentation of how to use them from CodeCentral at www.borland.com/CodeCentral.

Granting permission to access and launch the application server

Topic groups See also

Requests from the InternetExpress application appear to the application server as originating from a guest account with the name IUSR_computername, where computername is the name of the system running the Web application. By default, this account does not have access or launch permission for the application server. If you try to use the Web application without granting these permissions, when the Web browser tries to load the requested page it times out with EOLE_ACCESS_ERROR.

Note: Because the application server runs under this guest account, it can't be shut down by other accounts.

To grant the Web application access and launch permissions, run DCOMCnfg.exe, which is located in the System32 directory of the machine that runs the application server. The following steps describe how to configure your application server:

1. When you run DCOMCnfg, select your application server in the list of applications on the Applications page.
2. Click the Properties button. When the dialog changes, select the Security page.
3. Select Use Custom Access Permissions, and press the Edit button. Add the name IUSR_computername to the list of accounts with access permission, where computername is the name of the machine that runs the Web application.
4. Select Use Custom Launch Permissions, and press the Edit button. Add IUSR_computername to this list as well.
5. Click the Apply button.

Using an XML broker

Topic groups See also

The XML broker serves two major functions:

- It fetches XML data packets from the application server and makes them available to the Web Items that generate HTML for the InternetExpress application.
- It receives updates in the form of XML delta packets from browsers and applies them to the application server.

Fetching XML data packets

Before the XML broker can supply XML data packets to the components that generate HTML pages, it must fetch them from the application server. To do this, it uses the *IAppServer* interface of the application server, which it acquires through a connection component. You must set the following properties so that the XML producer can use the application server's *IAppServer* interface:

- Set the *RemoteServer* property to the connection component that establishes the connection to the application server and gets its *IAppServer* interface. At design time, you can select this value from a drop-down list in the object inspector.
- Set the *ProviderName* property to the name of the provider component on the application server that represents the dataset for which you want XML data packets. This provider both supplies XML data packets and applies updates from XML delta packets. At design time, if the *RemoteServer* property is set and the connection component has an active connection, the Object Inspector displays a list of available providers. (If you are using a DCOM connection the application server must also be registered on the client machine).

Two properties let you indicate what you want to include in data packets:

- You can limit the number of records that are added to the data packet by setting the *MaxRecords* property. This is especially important for large datasets because InternetExpress applications send the entire data packet to client Web browsers. If the data packet is too large, the download time can become prohibitively long.
- If the provider on the application server represents a query or stored procedure, you may want to provide parameter values before obtaining an XML data packet. You can supply these parameter values using the *Params* property.

The components that generate HTML and javascript for the InternetExpress application automatically use the XML broker's XML data packet once you set their *XMLBroker* property. To obtain the XML data packet directly in code, use the *RequestRecords* method.

Note: When the XML broker supplies a data packet to another component (or when you call *RequestRecords*), it receives an *OnRequestRecords* event. You can use this event to supply your own XML string instead of the data packet from the application server. For example, you could fetch the XML data packet from the application server using *GetXMLRecords* and then edit it before supplying it to the emerging Web page.

Applying updates from XML delta packets

When you add the XML broker to the Web module (or data module containing a *TWebDispatcher*), it automatically registers itself with the Web dispatcher as an auto-dispatching object. This means that, unlike other components, you do not need to create an action item for the XML broker in order for it to respond to update messages from a Web browser. These messages contain XML delta packets that should be applied to the application server. Typically, they originate from a button that you create on one of the HTML pages produced by the Web client application.

So that the dispatcher can recognize messages for the XML broker, you must describe them using the *WebDispatch* property. Set the *PathInfo* property to the path portion of the URL to which messages for the XML broker are sent. Set *MethodType* to the value of the method header of update messages addressed to that URL (typically *mtPost*). If you want to respond to all messages with the specified path, set *MethodType* to *mtAny*. If you don't want the XML broker to respond directly to update messages (for example, if you want to handle them explicitly using an action item), set the *Enabled* property to *False*. For more information on how the Web dispatcher determines which component handles messages from the Web browser, see Dispatching request messages.

When the dispatcher passes an update message on to the XML broker, it passes the updates on to the application server and, if there are update errors, receives an XML delta packet describing all update errors. Finally, it sends a response message back to the browser, which either redirects the browser to the same page that generated the XML delta packet or sends it some new content.

A number of events allow you to insert custom processing at all steps of this update process:

1. When the dispatcher first passes the update message to the XML broker, it receives a *BeforeDispatch* event, where you can preprocess the request or even handle it entirely. This event allows the XML broker to handle messages other than update messages.
2. If the *BeforeDispatch* event handler does not handle the message, the XML broker receives an *OnRequestUpdate* event, where you can apply the updates yourself rather than using the default processing.
3. If the *OnRequestUpdate* event handler does not handle the request, the XML broker applies the updates and receives a delta packet containing any update errors.
4. If there are no update errors, the XML broker receives an *OnGetResponse* event, where you can create a response message that indicates the updates were successfully applied or sends refreshed data to the browser. If the *OnGetResponse* event handler does not complete the response (does not set the *Handled* parameter to *True*), the XML broker sends a response that redirects the browser back to the document that generated the delta packet.
5. If there are update errors, the XML broker receives an *OnGetErrorResponse* event instead. You can use this event to try to resolve update errors or to generate a Web page that describes them to the end user. If the *OnGetErrorResponse* event handler does not complete the response (does not set the *Handled* parameter to *True*), the XML broker calls on a special content producer called the *ReconcileProducer* to generate the content of the response message.
6. Finally, the XML broker receives an *AfterDispatch* event, where you can perform any final actions before sending a response back to the Web browser.

Creating Web pages with a MIDAS page producer

Topic groups See also

Each MIDAS page producer generates an HTML document that appears in the browsers of your application's clients. If your application includes several separate Web documents, use a separate MIDAS page producer for each of them.

The MIDAS page producer is a special page producer component. As with other page producers, you can assign it as the *Producer* property of an action item or call it explicitly from an *OnAction* event handler. For more information about using content producers with action items, see Responding to request messages with action items.

Unlike most page producers, the MIDAS page producer has a default template as the value of its *HTMLDoc* property. This template contains a set of HTML-transparent tags that the MIDAS page producer uses to assemble an HTML document (with embedded javascript and XML) including content produced by other components. Before it can translate all of the HTML-transparent tags and assemble this document, you must indicate the location of the javascript libraries used for the embedded javascript on the page. This location is specified by setting the *IncludePathURL* property.

You can specify the components that generate parts of the Web page using the Web page editor. Display the Web page editor by double-clicking the Web page component or clicking the ellipsis button next to the *WebPageItems* property in the Object Inspector.

The components you add in the Web page editor generate the HTML that replaces one of the HTML-transparent tags in the MIDAS page producer's default template. These components become the value of the *WebPageItems* property. After adding the components in the order you want them, you can customize the template to add your own HTML or change the default tags.

Using the Web page editor

Topic groups See also

The Web page editor lets you add Web items to your MIDAS page producer and view the resulting HTML page. Display the Web page editor by double-clicking on a MIDAS page producer component.

Note: You must have Internet Explorer 4 or better installed to use the Web page editor.

The top of the Web page editor displays the Web items that generate the HTML document. These Web items are nested, where each type of Web item assembles the HTML generated by its subitems. Different types of items can contain different subitems. On the left, a tree view displays all of the Web items, indicating how they are nested. On the right, you can see the Web items included by the currently selected item. When you select a component in the top of the Web page editor, you can set its properties using the Object Inspector.

Click the New Item button to add a subitem to the currently selected item. The Add Web Component dialog lists only those items that can be added to the currently selected item.

The MIDAS page producer can contain one of two types of item, each of which generates an HTML form:

- *TDataForm*, which generates an HTML form for displaying data and the controls that manipulate that data or submit updates.

  Items you add to *TDataForm* display data in a multi-record grid (*TDataGrid*) or in a set of controls each of which represents a single field from a single record (*TFieldGroup*). In addition, you can add a set of buttons to navigate through data or post updates (*TDataNavigator*), or a button to apply updates back to the Web client (*TApplyUpdatesButton*). Each of these items contains subitems to represent individual fields or buttons. Finally, as with most Web items, you can add a layout grid (*TLayoutGroup*), that lets you customize the layout of any items it contains.
- *TQueryForm*, which generates an HTML form for displaying or reading application-defined values. For example, you can use this form for displaying and submitting parameter values.

  Items you add to *TQueryForm* display application-defined values(*TQueryFieldGroup*) or a set of buttons to submit or reset those values (*TQueryButtons*). Each of these items contains subitems to represent individual values or buttons. You can also add a layout grid to a query form, just as you can to a data form.

The bottom of the Web page editor displays the generated HTML code and lets you see what it looks like in a browser (IE4).

Setting Web item properties

Topic groups See also

The Web items that you add using the Web page editor are specialized components that generate HTML. Each Web item class is designed to produce a specific control or section of the final HTML document, but a common set of properties influences the appearance of the final HTML.

When a Web item represents information from the XML data packet (for example, when it generates a set of field or parameter display controls or a button that manipulates the data), the *XMLBroker* property associates the Web item with the XML broker that manages the data packet. You can further specify a dataset that is contained in a dataset field of that data packet using the *XMLDataSetField* property. If the Web item represents a specific field or parameter value, the Web item has a *FieldName* or *ParamName* property.

You can apply a style attribute to any Web item, thereby influencing the overall appearance of all the HTML it generates. Styles and style sheets are part of the HTML 4 standard. They allow an HTML document to define a set of display attributes that apply to a tag and everything in its scope. Web items offer a flexible selection of ways to use them:

- The simplest way to use styles is to define a style attribute directly on the Web item. To do this, use the *Style* property. The value of *Style* is simply the attribute definition portion of a standard HTML style definition, such as

```
color: red.
```

- You can also define a style sheet, that defines a set of style definitions. Each definition includes both a style selector (either the name of a tag to which the style always applies or a user-defined style name) and the attribute definition in curly braces:

```
H2 B  {color: red}
.MyStyle  {font-family: arial; font-weight: bold; font-size: 18px }
```

  The entire set of definitions is maintained by the MIDAS page producer as its *Styles* property. Each Web item can then reference the styles with user-defined names by setting its *StyleRule* property.
- You can also define a style sheet, that includes a set of style definitions. Each definition consists of a style selector (either the name of a tag to which the style applies or a user-defined style name) and an attribute definition in curly braces:

```
H2 B  {color: red}
.MyStyle  {font-family: arial; font-weight: bold; font-size: 18px }
```

  The entire set of definitions is maintained by the MIDAS page producer as its *Styles* property. Each Web item can then reference the styles with user-defined names by setting its *StyleRule* property.
- If you are sharing a style sheet with other applications, you can supply the style definitions as the value of the MIDAS page producer's *StylesFile* property instead of the *Styles* property. Individual Web items still reference styles using the *StyleRule* property.

Another common property of Web items is the *Custom* property. *Custom* includes a set of options that you add to the generated HTML tag. HTML defines a different set of options for each type of tag. The VCL reference for the *Custom* property of most Web items gives an example of possible options. For more information on possible options, use an HTML reference.

Customizing the MIDAS page producer template

Topic groups See also

The template of a MIDAS page producer is an HTML document with extra embedded tags that your application translates dynamically. Initially, the MIDAS page producer generates a default template as the value of the *HTMLDoc* property. This default template has the form

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<#INCLUDES> <#STYLES> <#WARNINGS> <#FORMS> <#SCRIPT>
</BODY>
</HTML>
```

The HTML-transparent tags in the default template are translated as follows:

<#INCLUDES> generates the statements that include the javascript libraries. These statements have the form

```
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmldom.js">
</SCRIPT>
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmldb.js">
</SCRIPT>
<SCRIPT language=Javascript type="text/javascript" SRC="IncludePathURL/xmlbind.js">
</SCRIPT>
```

<#STYLES> generates the statements that defines a style sheet from definitions listed in the *Styles* or *StylesFile* property of the MIDAS page producer.

<#WARNINGS> generates nothing at runtime. At design time, it adds warning messages for problems detected while generating the HTML document. You can see these messages in the Web page editor.

<#FORMS> generates the HTML produced by the components that you add in the Web page editor. The HTML from each component is listed in the order it appears in *WebPageItems*.

<#SCRIPT> generates a block of javascript declarations that are used in the HTML generated by the components added in the Web page editor.

You can replace the default template by changing the value of *HTMLDoc* or setting the *HTMLFile* property. The customized HTML template can include any of the HTML-transparent tags that make up the default template. The MIDAS page producer automatically translates these tags when you call the *Content* method. In addition, The MIDAS page producer automatically translates three additional tags:

<#BODYELEMENTS> is replaced by the same HTML as results from the 5 tags in the default template. It is useful when generating a template in an HTML editor when you want to use the default layout but add additional elements using the editor.

<#COMPONENT Name=WebComponentName> is replaced by the HTML that the component named *WebComponentName* generates. This component can be one of the components added in the Web page editor, or it can be any component that supports the *IWebContent* interface and has the same Owner as the MIDAS page producer.

<#DATAPACKET XMLBroker=BrokerName> is replaced with the XML data packet obtained from the XML broker specified by *BrokerName*. When, in the Web page editor, you see the HTML that the MIDAS page producer generates, you see this tag instead of the actual XML data packet.

In addition, the customized template can include any other HTML-transparent tags that you define. When the MIDAS page producer encounters a tag that is not one of the seven types it translates automatically, it generates an *OnHTMLTag* event, where you can write code to perform your own translations. For more information about HTML templates in general, see HTML templates.

Tip: The components that appear in the Web page editor generate static code. That is, unless the application server changes the metadata that appears in data packets, the HTML is always the same no matter when it is generated. You can avoid the overhead of generating this code dynamically at runtime in response to every request message by copying the generated HTML in the Web page editor and using it as a template. Because the Web page editor displays a

<#DATAPACKET> tag instead of the actual XML, using this as a template still allows your application to fetch data packets from the application server dynamically.

Using provider components

Topic groups See also

Provider components (*TDataSetProvider*) supply the mechanism by which client datasets obtain their data (unless they are using flat files). Providers are responsible for packaging data into data packets that are then sent to client datasets and applying updates received from client datasets. Usually they reside on an application server as part of a multi-tiered application, but they can appear as part of the same application as the client dataset (or XML broker). Providers work in conjunction with resolver components that handle the details of resolving data to the database or dataset.

Most of the work of a provider component happens automatically. You need not write any code on the provider to create a fully functioning application server. However, provider components include a number of events and properties that allow your application more direct control over what information is packaged for clients and how your application responds to client requests.

The following topics describe how to use a provider component to control the interaction with client applications:

- Determining the source of data
- Choosing how to apply updates
- Controlling what information is included in data packets
- Responding to client data requests
- Responding to client update requests
- Responding to client-generated events
- Handling server constraints

Determining the source of data

Topic groups See also

When you use a provider component, you must specify a dataset that it can use to get the data it assembles into data packets. To do this, set the *DataSet* property of the provider to the name of the dataset to use. At design time, select from available datasets in the *DataSet* property drop-down list in the Object Inspector.

TDataSetProvider can work with any dataset that supports the *IProviderSupport* interface. This interface is introduced by *TDataSet*, so it is available for all datasets. However, the *IProviderSupport* methods implemented in *TDataSet* are mostly stubs that don't do anything or that raise exceptions. Most of the dataset classes that ship with Delphi (BDE-enabled datasets, ADO-enabled datasets, Client datasets, and InterBase Express components) override these methods to implement the *IProviderSupport* interface in a more useful fashion.

Note: Because the provider relies on an interface belonging to the dataset, it has no specific dependencies on the data access mechanism (BDE, DBOLE, or some other mechanism). These dependencies all fall to the dataset that the provider uses.

Component writers that create their own custom descendants from *TDataSet* must override all appropriate *IProviderSupport* methods if their datasets are to work in an application server. If the provider only provides data packets on a read-only basis (that is, if it does not apply updates), the *IProviderSupport* methods implemented in *TDataSet* may be sufficient.

Choosing how to apply updates

Topic groups See also

By default, when *TDataSetProvider* components apply updates and resolve update errors, they communicate directly with the database server using dynamically generated SQL statements. This approach has the advantage that your server application does not need to merge updates twice (first to the dataset, and then to the remote server).

However, you may not always want to take this approach. For example, you may want to use some of the events on the dataset component. Alternately, the dataset you use may not support the use of SQL statements (for example if you are providing from a *TClientDataSet* component).

TDataSetProvider lets you decide whether to apply updates to the database server using SQL or to the source dataset by setting the *ResolveToDataSet* property. When this property is *True*, updates are applied to the dataset. When it is *False*, updates are applied directly to the underlying database server.

Controlling what information is included in data packets

Topic groups See also

There are a number of ways to control what information is included in data packets that are sent to and from the client. These include

- Specifying what fields appear in data packets.
- Setting Options that influence the data packets.
- Adding custom information to data packets.

Specifying what fields appear in data packets

Topic groups See also

To control what fields are included in data packets, create persistent fields on the dataset that the provider uses to build the packets. The provider then includes only these fields. Fields whose values are generated dynamically on the server (such as calculated fields or lookup fields) can be included, but appear to client datasets on the receiving end as static read-only fields.

If the client dataset will be editing the data and applying updates to the application server, you must include enough fields so that there are no duplicate records in the data packet. Otherwise, when the updates are applied, it is impossible to determine which record to update. If you do not want the client dataset to be able to see or use extra fields provided only to ensure uniqueness, set the *ProviderFlags* property for those fields to include *pfHidden*.

Note: Including enough fields to avoid duplicate records is also a consideration when using queries on the application server. The query should include enough fields so that records are unique, even if your application does not use all the fields.

Setting options that influence the data packets

Topic groups See also

The *Options* property of the provider component lets you specify when BLOBs or nested detail tables are sent, whether field display properties are included, what type of updates are allowed, and so on. The following table lists the possible values that can be included in *Options*.

Value	Meaning
poFetchBlobsOnDemand	BLOB field values are not included in the data packet. Instead, client applications must request these values on an as-needed basis. If the client dataset's *FetchOnDemand* property is *True*, the client requests these values automatically. Otherwise, the client application uses the client dataset's *FetchBlobs* method to retrieve BLOB data.
poFetchDetailsOnDemand	When the provider represents the master of a master/detail relationship, nested detail values are not included in the data packet. Instead, client applications request these on an as-needed basis. If the client dataset's *FetchOnDemand* property is *True*, the client requests these values automatically. Otherwise, the client application uses the client dataset's *FetchDetails* method to retrieve nested details.
poIncFieldProps	The data packet includes the following field properties (where applicable): *Alignment*, *DisplayLabel*, *DisplayWidth*, *Visible*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, *Currency*, *EditMask*, *DisplayValues.*
poCascadeDeletes	When the provider represents the master of a master/detail relationship, detail records are deleted by the server automatically when master records are deleted. To use this option, the database server must be set up to perform cascaded deletes as part of its referential integrity.
poCascadeUpdates	When the provider represents the master of a master/detail relationship, key values on detail tables are updated automatically when the corresponding values are changed in master records. To use this option, the database server must be set up to perform cascaded updates as part of its referential integrity.
poReadOnly	The client dataset can't apply updates to the provider.
poAllowMultiRecordUpdates	A single update can cause more than one record of the underlying database table to change. This can be the result of triggers, referential integrity, custom SQL statements, and so on. Note that if an error occurs, the event handlers provide access to the record that was updated, not the other records that change in consequence.
poDisableEdits	Clients can't modify existing data values. If the client tries to edit a field an exception is raised. (This does not affect the client's ability to insert or delete records).
poDisableInserts	Clients can't insert new records. If the client tries to insert a new record an exception is raised. (This does not affect the client's ability to delete records or modify existing data)
poDisableDeletes	Clients can't delete new records. If the client tries to delete a record an exception is raised. (This does not affect the client's ability to insert or modify records)
poNoReset	Clients can't specify that the provider should reposition the

	cursor on the first record before providing data.
poAutoRefresh	The provider refreshes the client dataset with current record values whenever it applies updates.
poPropogateChanges	Changes made by the server to updated records as part of the update process are sent back to the client and merged into the client dataset.
poAllowCommandText	The client can override the associated dataset's SQL text or the name of the table or stored procedure it represents.

Adding custom information to data packets

Topic groups See also

Providers can send application-defined information to the data packets using the *OnGetDataSetProperties* event. This information is encoded as an OleVariant, and stored under a name you specify. Client datasets in the client application can then retrieve the information using their *GetOptionalParam* method. You can also specify that the information be included in delta packets that the client dataset sends when updating records. In this case, the client application may never be aware of the information, but the server can send a round-trip message to itself.

When adding custom information in the *OnGetDataSetProperties* event, each individual attribute (sometimes called an “optional parameter”) is specified using a variant array that contains three elements: the name (a string), the value (a Variant), and a boolean flag indicating whether the information should be included in delta packets when the client applies updates. Multiple attributes can be added by creating a variant array of variant arrays. For example, the following *OnGetDataSetProperties* event handler sends two values, the time the data was provided and the total number of records in the source dataset. Only information about the time the data was provided is returned when clients apply updates:

```
procedure TMyDataModule1.Provider1GetDataSetProperties(Sender: TObject; DataSet:
TDataSet; out Properties: OleVariant);
begin
  Properties := VarArrayCreate([0,1], varVariant);
  Properties[0] := VarArrayOf(['TimeProvided', Now, True]);
  Properties[1] := VarArrayOf(['TableSize', DataSet.RecordCount, False]);
end;
```

When the client applies updates, the time the original records were provided can be read in the provider’s *OnUpdateData* event:

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TClientDataSet);
var
  WhenProvided: TDateTime;
begin
  WhenProvided := DataSet.GetOptionalParam('TimeProvided');
  ...
end;
```

Responding to client data requests

Topic groups See also

In most multi-tiered applications, client requests for data are handled automatically. A client dataset requests a data packet by calling *GetRecords* (indirectly, through the *IAppServer* interface). The provider responds automatically by fetching data from the associated dataset, creating a data packet, and sending the packet to the client.

The provider has the option of editing data after it has been assembled into a data packet but before the packet is sent to the client. For example, the provider might want to encrypt sensitive data before it is sent on to the client, or to remove records from the packet based on some criterion (such as the user's role in an MTS application).

To edit the data packet before sending it on to the client, write an *OnGetData* event handler. The data packet is provided as a parameter in the form of a client dataset. Using the methods of this client dataset, data can be edited before it is sent to the client.

As with all method calls that are made through the *IAppServer* interface, the provider has an opportunity to communicate persistent state information with the client application before and after the call to *GetRecords*. This communication takes place using the *BeforeGetRecords* and *AfterGetRecords* event handlers.

Responding to client update requests

Topic groups See also

A provider applies updates to database records based on a *Delta* data packet received from a client application. The client requests updates by calling the *ApplyUpdates* method (indirectly, through the *IAppServer* interface).

As with all method calls that are made through the *IAppServer* interface, the provider has an opportunity to communicate persistent state information with the client application before and after the call to *ApplyUpdates*. This communication takes place using the *BeforeApplyUpdates* and *AfterApplyUpdates* event handlers.

When a provider receives an update request, it generates an *OnUpdateData* event, where you can edit the Delta packet before it is written to the dataset or influence how updates are applied. After the *OnUpdateData* event, the provider uses its associated resolver component to write the changes to the database.

The resolver component performs the update on a record-by-record basis. Before the resolver applies each record, it generates a *BeforeUpdateRecord* event on the provider, which you can use to screen updates before they are applied. If an error occurs when updating a record, the resolver calls the provider's *OnUpdateError* event handler to resolve the error. Usually errors occur because the change violates a server constraint or the database record was changed by a different application subsequent to its retrieval by this client application, but prior to the client's request to apply updates.

Update errors can be processed by either the application server or the client. Application servers should handle all update errors that do not require user interaction to resolve. When the application server can't resolve an error condition, it temporarily stores a copy of the offending record. When record processing is complete, the application server returns a count of the errors it encountered to the client dataset, and copies the unresolved records into a results data packet that it passes back to the client for further reconciliation.

The event handlers for all provider events are passed the set of updates as a client dataset. If your event handler is only dealing with certain types of updates, you can filter the dataset on the update status of records so that your event handler does not need to sort through records it won't be using. To do this, set the *StatusFilter* property of the client dataset.

Note: Applications must supply extra support when the updates are directed at a dataset that does not represent a single table.

Editing delta packets before updating the database

Topic groups See also

Before the provider applies updates to the database, it generates an *OnUpdateData* event. The *OnUpdateData* event handler receives a copy of the *Delta* packet as a parameter. This is a client dataset.

In the *OnUpdateData* event handler, you can use any of the properties and methods of the client dataset to edit the *Delta* packet before it is written to the dataset. One particularly useful property is the *UpdateStatus* property. *UpdateStatus* indicates what type of modification the current record in the delta packet represents. It can have any of the values in the following table:

Value	Description
usUnmodified	Record contents have not been changed
usModified	Record contents have been changed
usInserted	Record has been inserted
usDeleted	Record has been deleted

For example, the following *OnUpdateData* event handler inserts the current date into every new record that is inserted into the database:

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TClientDataSet);
begin
  with DataSet do
  begin
    First;
    while notEof do
    begin
      if UpdateStatus = usInserted then
      begin
        Edit;
        FieldByName('DateCreated').AsDateTime := Date;
        Post;
      end;
      Next;
    end;
end;
```

Influencing how updates are applied

Topic groups See also

The *OnUpdateData* event gives your provider a chance to indicate how records in the delta packet are applied to the database.

By default, changes in the delta packet are written to the database using automatically generated SQL UPDATE, INSERT, or DELETE statements such as

```
UPDATE EMPLOYEES
  set EMPNO = 748, NAME = 'Smith', TITLE = 'Programmer 1', DEPT = 52
WHERE
  EMPNO = 748 and NAME = 'Smith' and TITLE = 'Programmer 1' and DEPT = 47
```

Unless you specify otherwise, all fields in the delta packet records are included in the UPDATE clause and in the WHERE clause. However, you may want to exclude some of these fields. One way to do this is to set the *UpdateMode* property of the provider. *UpdateMode* can be assigned any of the following values:

Value	Meaning
upWhereAll	All fields are used to locate fields (the WHERE clause).
upWhereChanged	Only key fields and fields that are changed are used to locate records.
upWhereOnly	Only key fields are used to locate records.

You might, however, want even more control. For example, with the previous statement, you might want to prevent the EMPNO field from being modified by leaving it out of the UPDATE clause and leave the TITLE and DEPT fields out of the WHERE clause to avoid update conflicts when other applications have modified the data. To specify the clauses where a specific field appears, use the *ProviderFlags* property. *ProviderFlags* is a set that can include any of the values in the following table

Value	Description
pfInWhere	The field does not appear in the WHERE clause of generated INSERT, DELETE, and UPDATE statements.
pfInUpdate	The field does not appear in the UPDATE clause of generated UPDATE statements.
pfInKey	The field is used in the WHERE clause of a generated SELECT statement that executes when update failures occur. This SELECT statement tries to locate the current value of modified or deleted records, or a record causing key violations when insertions fail.
pfHidden	The field is included in records to ensure uniqueness, but can't be seen or used on the client side.

Thus, the following *OnUpdateData* event handler excludes the EMPNO field from the UPDATE clause and the TITLE and DEPT fields from the WHERE clause:

```
procedure TMyDataModule1.Provider1UpdateData(Sender: TObject; DataSet:
TClientDataSet);
begin
  with DataSet do
  begin
    FieldByName('EMPNO').UpdateFlags := [ufInUpdate];
    FieldByName('TITLE').UpdateFlags := [ufInWhere];
    FieldByName('DEPT').UpdateFlags := [ufInWhere];
  end;
end;
```

Note: You can use the *UpdateFlags* property to influence how updates are applied even if you are updating to a dataset and not using dynamically generated SQL. These flags still determine which fields are used to locate records and which fields get updated.

Screening individual updates

Topic groups See also

Immediately before each update is applied, the provider receives a *BeforeUpdateRecord* event. You can use this event to edit records before they are applied, similar to the way you can use the *OnUpdateData* event to edit entire delta packets. For example, the provider does not compare BLOB fields (such as memos) when checking for update conflicts. If you want to check for update errors involving BLOB fields, you can use the *BeforeUpdateRecord* event.

In addition, you can use this event to apply updates yourself or to screen and reject updates. The *BeforeUpdateRecord* event handler lets you signal to the resolver that an update has been handled already and should not be applied. The resolver then skips that record, but does not count it as an update error. For example, this event provides a mechanism for applying updates to a stored procedure (which can't be updated automatically), allowing the provider to skip any automatic processing once the record is updated from within the event handler.

Resolving update errors on the provider

Topic groups See also

When an error condition arises as the application server tries to post a record in the delta packet, an *OnUpdateError* event occurs. If the application server can't resolve an update error, it temporarily stores a copy of the offending record. When record processing is complete, the application server returns a count of the errors it encountered to the client dataset, and copies the unresolved records into a results data packet that it passes back to the client for further reconciliation.

This mechanism lets you handle any update errors you can resolve mechanically on the application server, while still allowing user interaction on the client application to correct error conditions.

The *OnUpdateError* handler gets a copy of the record that could not be changed, an error code from the database, and an indication of whether the resolver was trying to insert, delete, or update the record. The problem record is passed back in a client dataset. You should never use the data navigation methods on this dataset. However, for each field in the dataset, you can use the *NewValue*, *OldValue*, and *CurValue* properties to determine the cause of the problem and make any modifications to resolve the update error. If the *OnUpdateError* event handler can correct the problem, it sets the *Response* parameter so that the corrected record is applied.

Applying updates to datasets that do not represent a single table

Topic groups See also

When a resolver component generates SQL statements that apply updates directly to a database server, it needs the name of the database table that contains the records. This can be handled automatically for many datasets such as *TTable* or "live" *TQuery* components.

Automatic updates are a problem however, if the resolver must apply updates to the data underlying a stored procedure with a result set or a multi-table query. This is because there is no easy way to obtain the name of the table to which updates should be applied.

If the query or stored procedure is a BDE-enabled dataset (*TQuery* or *TStoredProc*) and it has an associated update object, the provider will use the update object. However, if there is no update object, you can supply the table name programmatically in an *OnGetTableName* event handler. Once an event handler supplies the table name, the resolver component can generate appropriate SQL statements to apply updates.

Note: Supplying a table name only works if the target of the updates is a single database table (that is, only the records in one table need to be updated). If the update requires making changes to multiple underlying database tables, you must explicitly apply the updates in code using the *BeforeUpdateRecord* event of the provider. Once this event handler has applied an update, you can set the event handler's *Applied* parameter to *True* so that the resolver does not generate an error.

Responding to client-generated events

Topic groups See also

Provider components implement a general-purpose event that lets you create your own calls from clients directly to the provider. This is the *OnDataRequest* event.

OnDataRequest is not part of the normal functioning of the provider. It is simply a hook to allow your clients to communicate directly with providers on the application server. The event handler takes an OleVariant as an input parameter and returns an OleVariant. By using OleVariants, the interface is sufficiently general to accommodate almost any information you want to pass to or from the provider.

To generate an *OnDataRequest* event, the client application calls the *DataRequest* method of the client dataset.

Handling server constraints

Topic groups See also

Most relational database management systems implement constraints on their tables to enforce data integrity. A constraint is a rule that governs data values in tables and columns, or that governs data relationships across columns in different tables. For example, most SQL-92 compliant relational databases support the following constraints:

- NOT NULL, to guarantee that a value supplied to a column has a value.
- NOT NULL UNIQUE, to guarantee that column value has a value and does not duplicate any other value already in that column for another record.
- CHECK, to guarantee that a value supplied to a column falls within a certain range, or is one of a limited number of possible values.
- CONSTRAINT, a table-wide check constraint that applies to multiple columns.
- PRIMARY KEY, to designate one or more columns as the table's primary key for indexing purposes.
- FOREIGN KEY, to designate one or more columns in a table that reference another table.

Note: This list is not exclusive. Your database server may support some or all of these constraints in part or in whole, and may support additional constraints. For more information about supported constraints, see your server documentation.

Database server constraints obviously duplicate many kinds of data checks that traditional desktop database applications have managed in the past. You can take advantage of server constraints in your multi-tiered database applications without having to duplicate the constraints in application server or client application code.

The provider is working with a BDE-enabled dataset, its *Constraints* property enables you to replicate and apply server constraints to data passed to and received from client applications. When *Constraints* is *True* (the default), your server's constraints are replicated to clients and affect client attempts to update data.

Important: Before the application server can pass constraint information on to the client application, it must retrieve the constraints from the database server. To import database constraints from the server, use SQL explorer to import the database server's constraints and default expressions into the Data Dictionary. Constraints and default expressions in the Data Dictionary are automatically made available to BDE-enabled datasets in the application server.

There may be times when you do not want to apply server constraints to data sent to a client application. For example, a client application that receives data in packets and permits local updating of records prior to fetching more records may need to disable some server constraints that might be triggered because of the temporarily incomplete set of data. To prevent constraint replication from the application server to a client dataset, set *Constraints* to *False*. Note that client datasets can disable and enable constraints using the *DisableConstraints* and *EnableConstraints* methods. For more information about enabling and disabling constraints from the client dataset, see Handling constraints.

Managing database sessions

Topic groups See also

Both standalone, full client database applications and database application servers can communicate with databases through the Borland Database Engine (BDE). An application's database connections, drivers, cursors, queries, and so on are maintained within the context of one or more BDE sessions. Sessions isolate a set of database access operations, such as database connections, without the need to start another instance of the application.

In an application, you can manage BDE sessions using the TSession and TSessionList components. Each *TSession* component in an application encapsulates a single BDE session. All sessions within an application are managed by a single *TSessionList* component.

All database applications automatically include a session component, named *Session*, that encapsulates the default BDE session. Applications can declare, create, and manipulate additional session components as needed.

All database applications also automatically include a session list component, named *Sessions*, that enables the application to manage all of its session components.

This section describes the session and session list components and explains how to use them to control BDE sessions in your full client database applications and database application servers.

Note: The default session and session list components provide a widely applicable set of defaults that can be used as is by most applications. Only applications that use multiple sessions (for example, because of the need to run concurrent queries against a single database) may need to manipulate its session and session list components.

The following topics are discussed in this section:

- Working with a session component
- Managing multiple sessions
- Using a session component in data modules

Working with a session component

Topic groups See also

A session component provides global management for a group of database connections within an application. When you create a full client database application or an application server, your application automatically contains a session component, named *Session*. As you add database and dataset components to the application, they are automatically associated with this default session. It also controls access to password protected Paradox files, and it specifies directory locations for sharing Paradox files over a network. Applications can control database connections and access to Paradox files by using the properties, events, and methods of the session.

You can use the default session to control all database connections in an application. Alternatively, you can add additional session components at design time or create them dynamically at runtime to control a subset of database connections in an application.

Some applications require additional session components, such as applications that run concurrent queries against the same database. In this case, each concurrent query must run under its own session. Multi-threaded database applications also require multiple sessions. Applications that use multiple sessions must manage those sessions through the session list component, *Sessions*. For more information about managing multiple sessions see, Managing multiple sessions.

The following topics are discussed in this section:

- Using the default session
- Creating additional sessions
- Naming a session
- Activating a session
- Customizing session start-up
- Specifying default database connection behavior
- Creating, opening, and closing database connections
- Dropping temporary database connections
- Searching for a database connection
- Retrieving information about a session
- Working with BDE aliases
- Iterating through a session's database components
- Specifying Paradox directory locations
- Working with password-protected Paradox and dBase tables

Using the default session

Topic groups See also

All database applications automatically include a default session. Delphi creates a default session component named *Session* for a database application each time it runs (note that its *SessionName* is “Default”). The default session provides global control over all database components not associated with another session, whether they are temporary (created by the session at runtime when a dataset is opened that is not associated with a database component you create) or persistent (explicitly created by your application). The default session is not visible in your data module or form at design time, but you can access its properties and methods in your code at runtime.

When you create a database component, it is automatically associated with the default session. You need only associate a database component with an explicitly named session if the component performs a simultaneous query against a database already opened by the default session. When creating a multi-threaded database application, you must create one additional session to handle each additional thread.

To use the default session, you need write no code unless your application must

- Modify the properties of the session, such as specifying when database connections for automatically generated database components should automatically be kept or dropped.
- Respond to session events, such as when the application attempts to access a password-protected Paradox file.
- Execute a session’s methods, such as opening or closing a database in response to user-initiated actions.
- Set the *NetFileDir* property to access Paradox tables on a network and set the *PrivateDir* property to a local hard drive to speed performance.

Whether you add database components to an application at design time or create them dynamically at runtime, they are automatically associated with the default session unless you specifically assign them to a different session. If your application opens a dataset that is not associated with a database component, Delphi automatically

- Creates a database component for it at runtime.
- Associates the database component with the default session.
- Initializes some of the database component’s key properties based on the default session’s properties.

Among the most important of these properties is *KeepConnections*, which determines when database connections are maintained or dropped by an application. For more information about *KeepConnections*, see Specifying default database connection behavior.

Creating additional sessions

Topic groups See also Example

You can create sessions to supplement the default session. At design time, you can place additional sessions on a data module (or form), set their properties in the Object Inspector, write event handlers for them, and write code that calls their methods. You can also create sessions, set their properties, and call their methods at runtime. This section describes how to create and delete sessions at runtime.

Note: Keep in mind that creating additional sessions is optional unless an application runs concurrent queries against a database or the application is multi-threaded.

To enable dynamic creation of a session component at runtime, follow these steps:

1. Declare a pointer to a *TSession* variable.
2. Instantiate a new session by calling the *Create* constructor. The constructor sets up an empty list of database components for the session, sets up an empty list of BDE callbacks for the session, sets the *KeepConnections* property to *True*, and adds the session to the list of sessions maintained by the application's session list component.
3. Set the *SessionName* property for the new session to a unique name. This property is used to associate database components with the session. For more information about the *SessionName* property, see Naming a session.
4. Activate the session and optionally adjust its properties.

Note: Never delete the default session.

You can also manage creating and opening of sessions using the *OpenSession* method of *TSessionList*. Using *OpenSession* is safer than calling *Create*, because *OpenSession* only creates a session if it does not already exist. For more information about using *OpenSession*, see Managing multiple sessions..

Example: Creating a session

The following code creates a new session component, assigns it a name, and opens the session for database operations that follow (not shown here). After use, it is destroyed with a call to the *Free* method.

```
var
  SecondSession: TSession;
begin
  SecondSession := TSession.Create;
  with SecondSession do
    try
      SessionName := 'SecondSession';
      KeepConnections := False;
      Open;
      ...
    finally
      SecondSession.Free;
    end;
end;
```

Naming a session

Topic groups See also Example

A session's *SessionName* property is used to name the session so that you can associate databases and datasets with it. For the default session, *SessionName* is "Default," For each additional session component you create, you must set its *SessionName* property to a unique value.

Database and dataset components have *SessionName* properties that correspond to the *SessionName* property of a session component. If you leave the *SessionName* property blank for a database or dataset component it is automatically associated with the default session. You can also set *SessionName* for a database or dataset component to a name that corresponds to the *SessionName* of a session component you create.

For more information about using the *TSessionList* component—and *Sessions* in particular—to control multiple sessions, see Managing multiple sessions.

Example: Naming a session

The following code uses the *OpenSession* method of the default *TSessionList* component, *Sessions*, to open a new session component, sets its *SessionName* to “InterBaseSession,” activate the session, and associate an existing database component *Database1* with that session:

```
var
  IBSession: TSession;
  ...
begin
  IBSession := Sessions.OpenSession('InterBaseSession');
  Database1.SessionName := 'InterBaseSession';
end;
```

Activating a session

Topic groups See also

Active is a Boolean property that determines if database and dataset components associated with a session are open. You can use this property to read the current state of a session's database and dataset connections, or to change it.

To determine the current state of a session, check *Active*. If *Active* is *False* (the default), all databases and datasets associated with the session are closed. If *True*, databases and datasets are open.

Setting *Active* to *True* triggers a session's *OnStartup* event, sets the *NetFileDir* and *PrivateDir* properties if they are assigned values, and sets the *ConfigMode* property. You can write an *OnStartup* event handler to perform other specific database start-up activities. For more information about *OnStartup*, see Working with password-protected Paradox and dBase tables. The *NetFileDir* and *PrivateDir* properties are only used for connecting to Paradox tables. For more information about them, see Specifying the control file location and Specifying a temporary files location. *ConfigMode* determines how the BDE handles BDE aliases created within the context of the session. For more information about *ConfigMode*, see Specifying alias visibility. To open database components within a session, see Creating, opening, and closing database connections.

After you activate a session, you can open its database connections by calling the *OpenDatabase* method.

For session components you place in a data module or form, setting *Active* to *False* when there are open databases or datasets closes them. At runtime, closing databases and datasets may invoke events associated with them.

Note: You cannot set *Active* to *False* for the default session at design time. While you can close the default session at runtime, it is not recommended.

For session components you create, use the Object Inspector to set *Active* to *False* at design time to disable all database access for a session with a single property change. You might want to do this if, during application design, you do not want to receive exceptions because a remote database is temporarily unavailable.

You can also use a session's *Open* and *Close* methods to activate or deactivate sessions other than the default session at runtime. For example, the following single line of code closes all open databases and datasets for a session:

```
Session1.Close;
```

This code sets Session1's *Active* property to *False*. When a session's *Active* property is *False*, any subsequent attempt by the application to open a database or dataset resets *Active* to *True* and calls the session's *OnStartup* event handler if it exists. You can also explicitly code session reactivation at runtime. The following code reactivates *Session1*:

```
Session1.Open;
```

Note: If a session is active you can also open and close individual database connections. For more information, see Closing a single database connection.

Customizing session start-up

Topic groups See also

You can customize a session's start-up behavior by writing an *OnStartup* event handler for it. *OnStartup* is triggered when a session is activated. A session is activated when it is first created, and subsequently, whenever its *Active* property is changed to *True* from *False* (for example, when a database or dataset is associated with a session is opened and there are currently no other open databases or datasets).

Specifying default database connection behavior

Topic groups See also

KeepConnections provides the default value for the *KeepConnection* property of temporary database components created at runtime. *KeepConnection* specifies what happens to a database connection established for a database component when all its datasets are closed. If *True* (the default), a constant, or *persistent*, database connection is maintained even if no dataset is active. If *False*, a database connection is dropped as soon as all its datasets are closed.

Note: Connection persistence for a database component you explicitly place in a data module or form is controlled by that database component's *KeepConnection* property. If set differently, *KeepConnection* for a database component always overrides the *KeepConnections* property of the session. For more information about controlling individual database connections within a session, see Creating, opening, and closing database connections.

KeepConnections should always be set to *True* for applications that frequently open and close all datasets associated with a database on a remote server. This setting reduces network traffic and speeds data access because it means that a connection need only be opened and closed once during the lifetime of the session. Otherwise, every time the application closes or reestablishes a connection, it incurs the overhead of attaching and detaching the database.

Note: Even when *KeepConnections* is *True* for a session, you can close inactive database connections for all temporary database components, and then free the temporary database components by calling the *DropConnections* method. For more information about *DropConnections*, see Dropping temporary database connections.

For example, the following code drops inactive connections and frees all temporary database components for the default session:

```
Session.DropConnections;
```

Creating, opening, and closing database connections

Topic groups See also

To open a database connection within a session, call the *OpenDatabase* method. *OpenDatabase* takes one parameter, the name of the database to open. This name is a BDE alias or the name of a database component. For Paradox or dBASE, the name can also be a fully qualified path name. For example, the following statement uses the default session and attempts to open a database connection for the database pointed to by the DBDEMOS alias:

```
var
 DBDemosDatabase: TDatabase;
begin
 DBDemosDatabase := Session.OpenDatabase('DBDEMOS');
 ...
```

OpenDatabase makes its own session active if it is not already active, and then determines if the specified database name matches the *DatabaseName* property of any database components for the session. If the name does not match an existing database component, *OpenDatabase* creates a temporary database component using the specified name. Each call to *OpenDatabase* increments a reference count for the database by 1. As long as this reference count remains greater than 0, the database is open. Finally, *OpenDatabase* calls the *Open* method of the database component to connect to the server.

The following topics discuss closing the database connection:

- Closing a single database connection
- Closing all database connections

Closing a single database connection

Topic groups See also

You can close an individual database connection with the *CloseDatabase* method, or close all connections for a session at once with the *Close* method. When you call *CloseDatabase*, a reference count for the database is decremented by 1. When the reference count for the database is 0, the database is closed and freed. *CloseDatabase* takes one parameter, the database to close. For example, the following statement closes the database connection opened in the example in the previous section:

```
Session.CloseDatabase(DBDemosDatabase);
```

If the specified database name is associated with a temporary database component, and the session's *KeepConnections* property is *False*, the temporary database component is freed, effectively closing the connection.

Note: If *KeepConnections* is *False* temporary database components are closed and freed automatically when the last dataset associated with the database component is closed. An application can always call *CloseDatabase* prior to that time to force closure. To free temporary database components when *KeepConnections* is *True*, call the database component's *Close* method, and then call the session's *DropConnections* method.

If the database component is persistent (meaning that the application specifically declares the component and instantiates it), and the session's *KeepConnections* property is *False*, *CloseDatabase* calls the database component's *Close* method to close the connection.

Note: Calling *CloseDatabase* for a persistent database component does not actually close the connection. To close the connection, call the database component's *Close* method directly.

Closing all database connections

Topic groups See also

You can close all database connections in two ways:

- Set the *Active* property for the session to *False*.
- Call the *Close* method for the session.

When you set *Active* to *False*, Delphi automatically calls the *Close* method. *Close* disconnects from all active databases by freeing temporary database components and calling each persistent database component's *Close* method. Finally, *Close* sets the session's BDE handle to **nil**.

Dropping temporary database connections

Topic groups See also

If the *KeepConnections* property for a session is *True* (the default), then database connections for temporary database components are maintained even if all the datasets used by the component are closed. You can eliminate these connections and free all inactive temporary database components for a session by calling the *DropConnections* method. For example, the following code frees all inactive, temporary database components for the default session:

```
Session.DropConnections;
```

Temporary database components for which one or more datasets are active are not dropped or freed by this call. To free these components, call *Close*.

Searching for a database connection

Topic groups See also Example

Use a session's *FindDatabase* method to determine whether or not a specified database component is already associated with a session. *FindDatabase* takes one parameter, the name of the database to search for. This name is a BDE alias or database component name. For Paradox or dBASE, the name can also be a fully-qualified path name.

FindDatabase returns the database component if it finds a match. Otherwise it returns **nil**.

Example: Using FindDatabase

The following code searches the default session for a database component using the DBDEMOS alias, and if it is not found, creates one and opens it:

```
var
  DB: TDatabase;
begin
  DB := Session.FindDatabase('DBDEMOS');
  if (DB = nil) then                                { database doesn't exist for
session so,}
    DB := Session.OpenDatabase('DBDEMOS');                          { create and
open it}
  if Assigned(DB) and DB.Active then begin
    DB.StartTransaction;
    ...
  end;
end;
```

Retrieving information about a session

Topic groups See also

You can retrieve information about a session and its database components by using a session's informational methods. For example, one method retrieves the names of all aliases known to the session, and another method retrieves the names of tables associated with a specific database component used by the session. The following table summarizes the informational methods to a session component:

Method	**Purpose**
GetAliasDriverName	Retrieves the BDE driver for a specified alias of a database.
GetAliasNames	Retrieves the list of BDE aliases for a database.
GetAliasParams	Retrieves the list of parameters for a specified BDE alias of a database.
GetConfigParams	Retrieves specific configuration information from the BDE configuration file.
GetDatabaseNames	Retrieves the list of BDE aliases and the names of any *TDatabase* components currently in use.
GetDriverNames	Retrieves the names of all currently installed BDE drivers.
GetDriverParams	Retrieves the list of parameters for a specified BDE driver.
GetStoredProcNames	Retrieves the names of all stored procedures for a specified database.
GetTableNames	Retrieves the names of all tables matching a specified pattern for a specified database.

Except for *GetAliasDriverName*, these methods return a set of values into a string list declared and maintained by your application. (*GetAliasDriverName* returns a single string, the name of the current BDE driver for a particular database component used by the session.)

For example, the following code retrieves the names of all database components and aliases known to the default session:

```
var
  List: TStringList;
begin
  List := TStringList.Create;
  try
    Session.GetDatabaseNames(List);
    ...
  finally
    List.Free;
  end;
end;
```

For a complete description of a session's informational methods, see TSession.

Working with BDE aliases

Topic groups See also

Because a session usually encapsulates a series of database connections, one property and many of a session component's methods work with BDE aliases. Each database component associated with a session has a BDE alias (although optionally a fully-qualified path name may be substituted for an alias when accessing Paradox and dBASE tables. BDE aliases and the associated TSession methods have three main uses:

- Determining visibility
- Retrieving alias and driver information
- Creating, modifying, and deleting aliases

The following sections describe these functional areas.

- Specifying alias visibility
- Making session aliases visible to other sessions and applications
- Determining known aliases, drivers, and parameters
- Creating, modifying, and deleting aliases

Specifying alias visibility

Topic groups See also

A session's *ConfigMode* property determines what BDE aliases are visible to the session. *ConfigMode* is a set that describes which types of sessions are visible. The default setting is *cmAll*, which translates into the set [*cfmVirtual*, *cfmPersistent*]. If *ConfigMode* is *cmAll*, a session can see all aliases created within the session, all aliases in the BDE configuration file on a user's system, and all aliases that the BDE maintains in memory.

The main purpose of *ConfigMode* is to enable an application to specify and restrict alias visibility at the session level. For example, setting *ConfigMode* to *cfmSession* restricts a session's view of aliases to those created within the session. All other aliases in the BDE configuration file and in memory are not available.

For a full description of *ConfigMode* and its settings, see ConfigMode.

Making session aliases visible to other sessions and applications

Topic groups See also

When an alias is created during a session, the BDE stores a copy of the alias in memory. By default this copy is local only to the session in which it is created. Another session in the same application can only see the alias if its *ConfigMode* property is *cmAll* or *cfmPersistent*.

To make an alias available to all sessions and to other applications, use the session's *SaveConfigFile* method. *SaveConfigFile* writes aliases in memory to the BDE configuration file where they can be read and used by other BDE-enabled applications.

Determining known aliases, drivers, and parameters

Topic groups See also

Five methods for session components enable an application to retrieve information about BDE aliases, including parameter information and driver information. They are:

- *GetAliasNames*, to list the aliases to which a session has access.
- *GetAliasParams*, to list the parameters for a specified alias.
- *GetAliasDriverName*, to return a string containing the name of the BDE driver used by the alias.
- *GetDriverNames*, to return a list of all BDE drivers available to the session.
- *GetDriverParams*, to return driver parameters for a specified driver.

For more information about using a session's informational methods, see Retrieving information about a session.. For more information about BDE aliases, parameters, and drivers, see the online BDE help file, BDE32.HLP.

Creating, modifying, and deleting aliases

Topic groups See also Example

A session can create, modify, and delete aliases during its lifetime. The *AddAlias* method creates a new BDE alias for an SQL database server. *AddStandardAlias* creates a new BDE alias for Paradox, dBASE, or ASCII tables.

AddAlias takes three parameters: a string containing a name for the alias, a string that specifies the SQL Links driver to use, and a string list populated with parameters for the alias. For more information about *AddAlias*, see TSession.AddAlias. For more information about BDE aliases and the SQL Links drivers, see the BDE online help, BDE32.HLP.

AddStandardAlias takes three string parameters: the name for the alias, the fully-qualified path to the Paradox or dBASE table to access, and the name of the default driver to use when attempting to open a table that does not have an extension. For more information about *AddStandardAlias*, see TSession.AddStandardAlias. For more information about BDE aliases, see the BDE online help, BDE32.HLP.

Note: When you add an alias to a session, it is only available to this session and any other sessions with *cfmPersistent* included in the *ConfigMode* property. To make a newly created alias available to all applications, call *SaveConfigFile* after creating the alias. For more information about *ConfigMode*, see Working with BDE aliases..

After you create an alias, you can make changes to its parameters by calling *ModifyAlias*. *ModifyAlias* takes two parameters: the name of the alias to modify and a string list containing the parameters to change and their values.

To delete an alias previously created in a session, call the *DeleteAlias* method. *DeleteAlias* takes one parameter, the name of the alias to delete. *DeleteAlias* makes an alias unavailable to the session.

Note: *DeleteAlias* does not remove an alias from the BDE configuration file if the alias was written to the file by a previous call to *SaveConfigFile*. To remove the alias from the configuration file after calling *DeleteAlias*, call *SaveConfigFile* again.

Example: Adding and modifying aliases

The following statements use *AddAlias* to add a new alias for accessing an InterBase server to the default session:

```
var
  AliasParams: TStringList;
begin
  AliasParams := TStringList.Create;
  try
    with AliasParams do begin
      Add('OPEN MODE=READ');
      Add('USER NAME=TOMSTOPPARD');
      Add('SERVER NAME=ANIMALS:/CATS/PEDIGREE.GDB');
    end;
    Session.AddAlias('CATS', 'INTRBASE', AliasParams);
    ...
  finally
    AliasParams.Free;
  end;
end;
```

The following statement uses *AddStandardAlias* to create a new alias for accessing a Paradox table:

```
AddStandardAlias('MYDBDEMOS', 'C:\TESTING\DEMOS\', 'Paradox');
```

The following statements use *ModifyAlias* to change the OPEN MODE parameter for the CATS alias to READ/WRITE in the default session:

```
var
  List: TStringList;
begin
  List := TStringList.Create;
  with List do begin
    Clear;
    Add('OPEN MODE=READ/WRITE');
  end;
  Session.ModifyAlias('CATS', List);
  List.Free;
  ...
```

Iterating through a session's database components

Topic groups See also Example

Two session component properties, *Databases* and *DatabaseCount*, enable you to cycle through all the database components associated with a session.

Databases is an array of all currently active database components associated with a session. Used with the *DatabaseCount* property, *Databases* can be used to iterate through all active database components to perform a selective or universal action.

DatabaseCount is an Integer property that indicates the number of currently active databases associated with a session. As connections are opened or closed during a session's life-span, this number can change. For example, if a session's *KeepConnections* property is *False* and all database components are created as needed at runtime, each time a unique database is opened, *DatabaseCount* increases by one. Each time a unique database is closed, *DatabaseCount* decreases by one. If *DatabaseCount* is zero, there are no currently active database components for the session.

DatabaseCount is typically used with the *Databases* property to perform actions common to active database components.

Example: Iterating through a session's database components

The following example code sets the *KeepConnection* property of each active database in the default session to *True*:

```
var
  MaxDbCount: Integer;
begin
  with Session do
    if (DatabaseCount > 0) then
      for MaxDbCount := 0 to (DatabaseCount - 1) do
        Databases[MaxDbCount].KeepConnection := True;
end;
```

Specifying Paradox directory locations

Topic groups See also

Two session component properties, *NetFileDir* and *PrivateDir*, are specific to applications that work with Paradox tables. *NetFileDir* specifies the directory that contains the Paradox network control file, PDOXUSRS.NET. This file governs sharing of Paradox tables on network drives. All applications that need to share Paradox tables must specify the same directory for the network control file (typically a directory on a network file server).

PrivateDir specifies the directory for storing temporary table processing files, such as those generated by the BDE to handle local SQL statements.

The following topics are discussed in this section:

- Specifying the control file location
- Specifying a temporary files location

Specifying the control file location

Topic groups See also

Delphi derives a value for *NetFileDir* from the Borland Database Engine (BDE) configuration file for a given database alias. If you set *NetFileDir* yourself, the value you supply overrides the BDE configuration setting, so be sure to validate the new value.

At design time, you can specify a value for *NetFileDir* in the Object Inspector. You can also set or change *NetFileDir* in code at runtime. The following code sets *NetFileDir* for the default session to the location of the directory from which your application runs:

```
Session.NetFileDir := ExtractFilePath(Application.EXEName);
```

Note: *NetFileDir* can only be changed when an application does not have any open Paradox files. If you change *NetFileDir* at runtime, verify that it points to a valid network directory that is shared by your network users.

Specifying a temporary files location

Topic groups See also

If no value is specified for the *PrivateDir* property, the BDE automatically uses the current directory at the time it is initialized. If your application runs directly from a network file server, you can improve application performance at runtime by setting *PrivateDir* to a user's local hard drive before opening the database.

Note: Do not set *PrivateDir* at design time and then open the session in the IDE. Doing so generates a Directory is busy error when running your application from the IDE.

The following code changes the setting of the default session's *PrivateDir* property to a user's C:\TEMP directory:

```
Session.PrivateDir := 'C:\TEMP';
```

Important: Do not set *PrivateDir* to a root directory on a drive. Always specify a subdirectory.

Working with password-protected Paradox tables

Topic groups See also

A session component provides four methods and one event that are exclusively used to manage access to password-protected Paradox and dBase files. The methods are *AddPassword*, *GetPassword*, *RemoveAllPasswords*, and *RemovePassword*. The event is *OnPassword*. The *PasswordDialog* function is also available for adding and removing one or more passwords from a session.

Keys to working with password-protected Paradox tables include:

- Using the AddPassword method.
- Using the RemovePassword and RemoveAllPasswords methods.
- Using the GetPassword method and OnPassword event.

Using the AddPassword method

Topic groups

The *TSession.AddPassword* method provides an optional way for an application to provide a password for a session prior to opening an encrypted Paradox or dBase table that requires a password for access. *AddPassword* takes one parameter, a string containing the password to use. You can call *AddPassword* as many times as necessary to add passwords to access files protected with different passwords.

```
var
  Passwrd: String;
begin
  Passwrd := InputBox('Enter password', 'Password:', '');
  Session.AddPassword(Passwrd);
  try
    Table1.Open
  except
    ShowMessage('Could not open table!');
    Application.Terminate;
  end;
end;
```

Use of the *InputBox* function, above, is for demonstration purposes. In a real-world application, use password entry facilities that mask the password as it is entered, such as the *PasswordDialog* function or a custom form. On a custom password entry form, use a *TEdit* with the *PasswordChar* set to an asterisk ("*").

The Add button of the *PasswordDialog* function dialog has the same effect as the *AddPassword* method.

```
if PasswordDialog(Session) then
  Table1.Open
else
  ShowMessage('No password given, could not open table!');
end;
```

Note: You need to call *AddPassword* to specify one or more passwords (one at a time) to use when accessing password-protected files. If you do not, when your application attempts to open a password-protected table, a dialog box prompts the user for a password.

Using the RemovePassword and RemoveAllPasswords methods

Topic groups

TSession.RemovePassword deletes a previously added password from memory. *RemovePassword* takes one parameter, a string containing the password to delete.

```
Session.RemovePassword('secret');
```

TSession.RemoveAllPasswords deletes all previously added passwords from memory.

```
Session.RemoveAllPasswords;
```

Using the GetPassword method and OnPassword event

Topic groups Example

TSession.GetPassword triggers the *TSession.OnPassword* event for a session. The *OnPassword* event is called only when an application attempts to open a Paradox or dBase table for the first time and the BDE reports insufficient access rights. You can code an *OnPassword* event handler and provide a password to the BDE, or you can choose to use the default password handling, which displays a dialog box prompting the user for a password.

Example: Using OnPassword

In the example below, the procedure *Password* is designated as the handler for the *OnPassword* event by assigning the procedure's name to the *TSession.OnPassword* property.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Session.OnPassword := Password;
end;
```

In the *Password* procedure, the *InputBox* function is used to prompt the user for a password. The *TSession.AddPassword* method is used to programmatically supply the password entered in the dialog to the session.

```
procedure TForm1.Password(Sender: TObject; var Continue: Boolean);
var
  Passwrd: String;
begin
  Passwrd := InputBox('Enter password', 'Password:', '');
  Continue := (Passwrd > '');
  Session.AddPassword(Passwrd);
end;
```

Use of the *InputBox* function, above, is for demonstration purposes. In a real-world application, use password entry facilities that mask the password as it is entered, such as the *PasswordDialog* function or a custom form. On a custom password entry form, use a *TEdit* with the *PasswordChar* property set to an asterisk ("*").

The *Password* procedure is triggered by an attempt to open a password-protected table, as shown below. Exception handling can be used to accommodate a failed attempt to open the table. Even though the user is prompted for a password in the *OnPassword* event handler, the open attempt can still fail if they enter an invalid password or something else goes wrong.

```
procedure TForm1.OpenTableBtnClick(Sender: TObject);
const
  CRLF = #13 + #10;
begin
  try
    Table1.Open;                                { this line triggers the OnPassword
event }
  except
    on E:Exception do begin                             { exception if cannot open
table }
      ShowMessage('Error!' + CRLF +             { display error explaining what
happened }
        E.Message + CRLF +
        'Terminating application...');
      Application.Terminate;                                    { end the
application }
    end;
  end;
end;
```

Managing multiple sessions

Topic groups See also

If you create a single application that uses multiple threads to perform database operations, you must create one additional session for each thread. The Data Access page on the Component palette contains a session component that you can place in a data module or on a form at design time.

Important: When you place a session component, you must also set its *SessionName* property to a unique value so that it does not conflict with the default session's *SessionName* property.

Placing a session component at design time presupposes that the number of threads (and therefore sessions) required by the application at runtime is static. More likely, however, is that an application needs to create sessions dynamically. To create sessions dynamically, call the global function *Sessions.OpenSession* at runtime.

Sessions.OpenSession requires a single parameter, a name for the session that is unique across all session names for the application. The following code dynamically creates and activates a new session with a uniquely generated name:

```
Sessions.OpenSession('RunTimeSession' + IntToStr(Sessions.Count + 1));
```

This statement generates a unique name for a new session by retrieving the current number of sessions, and adding one to that value. Note that if you dynamically create and destroy sessions at runtime, this example code will not work as expected. Nevertheless, this example illustrates how to use the properties and methods of *Sessions* to manage multiple sessions.

Sessions is a variable of type *TSessionList* that is automatically instantiated for database applications. You use the properties and methods of *Sessions* to keep track of multiple sessions in a multi-threaded database application. The following table summarizes the properties and methods of the *TSessionList* component:

Property or Method	Purpose
Count	Returns the number of sessions, both active and inactive, in the sessions list.
FindSession	Searches the session list for a session with a specified name, and returns a pointer to the session component, or **nil** if there is no session with the specified name. If passed a blank session name, *FindSession* returns a pointer to the default session, *Session*.
GetSessionNames	Populates a string list with the names of all currently instantiated session components. This procedure always adds at least one string, "Default" for the default session (note that the default session's name is actually a blank string).
List	Returns the session component for a specified session name. If there is no session with the specified name, an exception is raised.
OpenSession	Creates and activates a new session or reactivates an existing session for a specified session name.
Sessions	Accesses the session list by ordinal value.

As an example of using *Sessions* properties and methods in a multi-threaded application, consider what happens when you want to open a database connection. To determine if a connection already exists, use the *Sessions* property to walk through each session in the sessions list, starting with the default session. For each session component, examine its *Databases* property to see if the database in question is open. If you discover that another thread is already using the desired database, examine the next session in the list.

If an existing thread is not using the database, then you can open the connection within that session.

If, on the other hand, all existing threads are using the database, you must open a new session in which to open another database connection.

If you are replicating a data module that contains a session in a multi-threaded application, where each thread contains its own copy of the data module, you can use the *AutoSessionName* property to make

sure that all datasets in the data module use the correct session. Setting *AutoSessionName* to *True* causes the session to generate its own unique name dynamically when it is created at runtime. It then assigns this name to every dataset in the data module, overriding any explicitly set session names. This ensures that each thread has its own session, and each dataset uses the session in its own data module.

Using a session component in data modules

Topic groups See also

You can safely place session components in data modules. If you put a data module that contains one or more session components into the Object Repository, however, make sure to set the *AutoSessionName* property to *True* to avoid namespace conflicts when users inherit from it.

Connecting to databases

Topic groups See also

When a Delphi application uses the Borland Database Engine (BDE) to connect to a database, that connection is encapsulated by a *TDatabase* component. A database component encapsulates the connection to a single database in the context of a BDE session. The following topics describe database components and how to manipulate database connections.

- Understanding persistent and temporary database components
- Controlling connections
- Understanding database and session component interactions
- Using database components in data modules

Database components are also used to manage transactions in BDE-based applications. For more information about using databases to manage transactions, see Using transactions.

Another use for database components is applying cached updates for related tables. For more information about using a database component to apply cached updates, see Applying cached updates with a database component method.

Understanding persistent and temporary database components

Topic groups See also

Each BDE-based database connection in an application is encapsulated by a database component whether or not you explicitly provide a database component at design time or create it dynamically at runtime. When an application attempts to connect to a database, it either uses an explicitly instantiated, or *persistent*, database component, or it generates a temporary database component that exists only for the lifetime of the connection.

Temporary database components are created as necessary for any datasets in a data module or form for which you do not create yourself. Temporary database components provide broad support for many typical desktop database applications without requiring you to handle the details of the database connection. For most client/server applications, however, you should create your own database components instead of relying on temporary ones. You gain greater control over your databases, including the ability to

- Create persistent database connections.
- Customize database server logins.
- Control transactions and specify transaction isolation levels.
- Create BDE aliases local to your application.

The following topics are discussed in this section:

- Using temporary database components
- Creating database components at design time
- Creating database components at runtime

Using temporary database components

Topic groups See also

Temporary database components are automatically generated as needed. For example, if you place a *TTable* component on a form, set its properties, and open the table without first placing and setting up a *TDatabase* component and associating the table component with it, Delphi creates a temporary database component for you behind the scenes.

Some key properties of temporary database components are determined by the session to which they belong. For example, the controlling session's *KeepConnections* property determines whether a database connection is maintained even if its associated datasets are closed (the default), or if the connections are dropped when all its datasets are closed. Similarly, the default *OnPassword*event for a session guarantees that when an application attempts to attach to a database on a server that requires a password, it displays a standard password prompt dialog box. Other properties of temporary database components provide standard login and transaction handling. For more information about sessions and session control over temporary database connections, see "Working with a session component".

The default properties created for temporary database components provide reasonable, general behaviors meant to cover a wide variety of situations. For complex, mission-critical client/server applications with many users and different requirements for database connections, however, you should create your own database components to tune each database connection to your application's needs.

Creating database components at design time

Topic groups See also

The Data Access page of the Component palette contains a database component you can place in a data module or form. The main advantages to creating a database component at design time are that you can set its initial properties and write *OnLogin* events for it. *OnLogin* offers you a chance to customize the handling of security on a database server when a database component first connects to the server. For more information about managing connection properties, see "Connecting to a database server." For more information about server security, see "Controlling server login."

Creating database components at runtime

Topic groups See also

You can create database components dynamically at runtime. An application might do this when the number of database components needed at runtime is unknown, and your application needs explicit control over the database connection. In fact, this is how Delphi itself creates temporary database components as needed at runtime. When you create a database component at runtime, you need to give it a unique name and to associate it with a session.

You create the component by calling the *TDatabase.Create* constructor. Given both a database name and a session name, the following function creates a database component at runtime, associates it with a specified session (creating a new session if necessary), and sets a few key database component properties:

```
function TDataForm.RunTimeDbCreate(const DatabaseName, SessionName: String):
TDatabase;
var
  TempDatabase: TDatabase;
begin
  TempDatabase := nil;
  try
    { If the session exists, make it active; if not, create a new session }
    Sessions.OpenSession(SessionName);
    with Sessions do
      with FindSession(SessionName) do begin
        Result := FindDatabase(DatabaseName);
        if (Result = nil) then begin
          { Create a new database component }
          TempDatabase := TDatabase.Create(Self);
          TempDatabase.DatabaseName := DatabaseName;
          TempDatabase.SessionName := SessionName;
          TempDatabase.KeepConnection := True;
        end;
        Result := OpenDatabase(DatabaseName);
      end;
  except
    TempDatabase.Free;
    raise;
  end;
end;
```

The following code fragment illustrates how you might call this function to create a database component for the default session at runtime:

```
var
  MyDatabase: array [1..10] of  TDatabase;
  MyDbCount: Integer;
begin
  { Initialize MyDbCount early on }
  MyDbCount := 1;
  ...
  { Later, create a database component at runtime }
  begin
    MyDatabase[MyDbCount] := RunTimeDbCreate('MyDb' + IntToStr(MyDbCount), '');
    Inc(MyDbCount);
  end;
  ...
end;
```

Controlling connections

Topic groups See also

Whether you create a database component at design time or runtime, you can use the properties, events, and methods of TDatabase to control and change its behavior in your applications. The following sections describe how to manipulate database components.

- Associating a database component with a session
- Specifying a BDE alias
- Setting BDE alias parameters
- Controlling server login
- Connecting to a database server
- Special considerations when connecting to a remote server
- Disconnecting from a database server
- Closing datasets without disconnecting from a server
- Iterating through a database component's datasets

Associating a database component with a session

Topic groups See also

All database components must be associated with a BDE session. Two database component properties, *Session* and *SessionName*, establish this association.

SessionName identifies the session alias with which to associate a database component. When you first create a database component at design time, *SessionName* is set to “Default”. Multi-threaded or reentrant BDE applications may have more than one session. At design time, you can pick a valid *SessionName* from the drop-down list in the Object Inspector. Session names in the list come from the *SessionName* properties of each session component in the application.

The *Session* property is a runtime, read-only property that points to the session component specified by the *SessionName* property. For example, if *SessionName* is blank or “Default”, then the *Session* property references the same *TSession* instance referenced by the global *Session* variable. *Session* enables applications to access the properties, methods, and events of a database component’s parent session component without knowing the session’s actual name. This is useful when a database component is assigned to a different session at runtime.

For more information about BDE sessions, see "Managing database sessions.”

Specifying a BDE alias

Topic groups See also

AliasName and *DriverName* are mutually exclusive BDE-specific properties. *AliasName* specifies the name of an existing BDE alias to use for the database component. The alias appears in subsequent drop-down lists for dataset components so that you can link them to a particular database component. If you specify *AliasName* for a database component, any value already assigned to *DriverName* is cleared because a driver name is always part of a BDE alias.

Note: You create and edit BDE aliases using the Database Explorer or the BDE Administration utility. For more information about creating and maintaining BDE aliases, see the online documentation for these utilities.

DatabaseName enables you to provide an application-specific name for a database component. The name you supply is in addition to *AliasName* or *DriverName*, and is local to your application. *DatabaseName* can be a BDE alias, or, for Paradox and dBASE files, a fully-qualified path name. Like *AliasName*, *DatabaseName* appears in subsequent drop-down lists for dataset components to enable you to link them to a database component.

DriverName is the name of a BDE driver. A driver name is one parameter in a BDE alias, but you may specify a driver name instead of an alias when you create a local BDE alias for a database component using the *DatabaseName* property. If you specify *DriverName*, any value already assigned to *AliasName* is cleared to avoid potential conflicts between the driver name you specify and the driver name that is part of the BDE alias identified in *AliasName*.

At design time, to specify a BDE alias, assign a BDE driver, or create a local BDE alias, double-click a database component to invoke the Database Properties editor.

You can enter a *DatabaseName* in the Name edit box in the properties editor. You can enter an existing BDE alias name in the Alias name combo box for the *Alias* property, or you can choose from existing aliases in the drop-down list. The Driver name combo box enables you to enter the name of an existing BDE driver for the *DriverName* property, or you can choose from existing driver names in the drop-down list.

Note: The Database Properties editor also enables you to view and set BDE connection parameters, and set the states of the *LoginPrompt* and *KeepConnection* properties. To work with connection parameters, see "Setting BDE alias parameters." To set the state of *LoginPrompt*, see "Controlling server login," and to set *KeepConnection* see "Connecting to a database server."

To set *DatabaseName*, *AliasName*, or *DriverName* at runtime, include the appropriate assignment statement in your code. For example, the following code uses the text from an edit box to create a local alias for the database component *Database1*:

```
Database1.DatabaseName := Edit1.Text;
```

Setting BDE alias parameters

Topic groups See also

At design time you can create or edit connection parameters in three ways:

- Use the Database Explorer or BDE Administration utility to create or modify BDE aliases, including parameters. For more information about these utilities, see their online Help files.
- Double-click the *Params* property in the Object Inspector to invoke the String List editor. To learn more about the String List editor, see Working with String lists.
- Double-click a database component in a data module or form to invoke the Database Properties editor.

All of these methods edit the *Params* property for the database component. *Params* is a string list containing the database connection parameters for the BDE alias associated with a database component. Some typical connection parameters include path statement, server name, schema caching size, language driver, and SQL query mode.

When you first invoke the Database Properties editor, the parameters for the BDE alias are not visible. To see the current settings, click Defaults. The current parameters are displayed in the Parameter overrides memo box. You can edit existing entries or add new ones. To clear existing parameters, click Clear. Changes you make take effect only when you click OK.

At runtime, an application can set alias parameters only by editing the *Params* property directly. For more information about parameters specific to using SQL Links drivers with the BDE, see your online SQL Links help file.

Controlling server login

Topic groups See also

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

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

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

- Let the default login dialog and processes handle the login. This is the default approach. Set the *LoginPrompt* property of the database component to *True* (the default). Your application displays the standard login dialog box when the server requests a user name and password.
- Supply the login information from the application, before the attempt to login. Assign values to the USER NAME and PASSWORD parameters of the database component through its *Params* property. Set the *LoginPrompt* to *False*, to prevent the default login dialog from appearing. The values in the Params property may be set at design-time through the Object Inspector or programmatically at runtime. For example:

```
with Database1 do begin
  Params.Values['USER NAME'] := 'SYSDBA';
  Params.Values['PASSWORD'] := 'masterkey';
  LoginPrompt := False;
  Connected := True;
end;
```

Important: Setting the values in the Params property at design-time causes the values to be embedded in the application's executable file and easy to find. This compromises server security, so it is not recommended. Values provided at runtime, such as entered by the end-user into a custom login dialog, do not present this situation and so do not compromise database security.

- Provide your own custom handling for the login event. Set the *LoginPrompt* property to *True* and write procedure to act as a handler for the *OnLogin* event for the database component. In the procedure used for the *OnLogin* event handler, set the login parameters. A copy of the database component's parameters is passed to the event handler in its *LoginParams* parameter. Assign values to the USER NAME and PASSWORD database parameters in this passed string list object. Provide what information is needed to successfully login to the database. For instance, the USER NAME may already be provided by a BDE alias and only the PASSWORD parameter needs to be supplied. Use the *Values* property of *TStrings* to set or change login parameters. Here is an example where the values for the USER NAME and PASSWORD database parameters are provided from memory **String** variables set elsewhere in the application:

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

On exit, *OnLogin* passes its *LoginParams* values back to *Params*, which is used to establish a connection.

Important: As is the case when hard-coding the password into the *Params* property, supplying the PASSWORD parameter's value in *LoginParams* from a value hard-coded into the application will compromise database security. The value for the PASSWORD parameter is a **String** and so can come from such sources as an encrypted value in the executable file or entered by the end-user at runtime through a login dialog of your own creation.

Important: The *OnLogin* even will not fire at all unless the *LoginPrompt* property is set to *True*. So having a *LoginPrompt* value of *False* and assigning a handler for the *OnLogin* event creates a situation where it is impossible to login to the database. The default dialog does not appear and the *OnLogin* event handler never executes.

Connecting to a database server

Topic groups See also

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

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

Setting *Connected* to *True* executes the *Open* method. *Open* verifies that the database specified by the *DatabaseName* or *Directory* properties exists, and the *OnLogin* event for the database component fires. If a procedure has been assigned as the handler for the *OnLogin* event and the *LoginPrompt* is set to True, this event handler is executed when the *OnLogin* event fires. Otherwise, the default login dialog box appears.

Note: When a database component is not connected to a server and an application attempts to open a dataset associated with the database component, the database component's *Open* method is first called to establish the connection. If the dataset is not associated with an existing database component, a temporary database component is created and used to establish the connection.

Once a database connection is established the connection is maintained as long as there is at least one active dataset. When there are no more active datasets, whether or not the connection is dropped depends on the setting of the database component's *KeepConnection* property.

KeepConnection determines if your application maintains a connection to a database even when all datasets associated with that database are closed. If *True*, a connection is maintained. For connections to remote database servers, or for applications that frequently open and close datasets, make sure *KeepConnection* is *True* to reduce network traffic and speed up your application. If *False* a connection is dropped when there are no active datasets using the database. If a dataset is later opened which uses the database, the connection must be reestablished and initialized.

Special considerations when connecting to a remote server

Topic groups See also

When you connect to a remote database server from an application, the application uses the BDE and the Borland SQL Links driver to establish the connection. (The BDE can communicate with an ODBC driver that you supply.) You need to configure the SQL Links or ODBC driver for your application prior to making the connection. SQL Links and ODBC parameters are stored in the *Params* property of a database component. For information about SQL Links parameters, see the online *SQL Links User's Guide*. To edit the *Params* property, see "Setting BDE alias parameters."

The following topics are discussed in this section:

- Working with network protocols
- Using ODBC

Working with network protocols

Topic groups See also

As part of configuring the appropriate SQL Links or ODBC driver, you may need to specify the network protocol used by the server, such as SPX/IPX or TCP/IP, depending on the driver's configuration options. In most cases, network protocol configuration is handled using a server's client setup software. For ODBC it may also be necessary to check the driver setup using the ODBC driver manager.

Establishing an initial connection between client and server can be problematic. The following troubleshooting checklist should be helpful if you encounter difficulties:

- Is your server's client-side connection properly configured?
- If you are using TCP/IP:
- Is your TCP/IP communications software installed? Is the proper WINSOCK.DLL installed?
- Is the server's IP address registered in the client's HOSTS file?
- Is the Domain Name Services (DNS) properly configured?
- Can you ping the server?
- Are the DLLs for your connection and database drivers in the search path?

For more troubleshooting information, see the online *SQL Links User's Guide* and your server documentation.

Using ODBC

Topic groups See also

An application can use ODBC data sources (for example, Btrieve). An ODBC driver connection requires

- A vendor-supplied ODBC driver.
- The Microsoft ODBC Driver Manager.
- The BDE Administration utility.

To set up a BDE alias for an ODBC driver connection, use the BDE Administration utility. For more information, see the BDE Administration utility's online help file.

Disconnecting from a database server

Topic groups See also

There are two ways to disconnect a server from a database component:

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

Setting *Connected* to *False* calls *Close*. *Close* closes all open datasets and disconnects from the server. For example, the following code closes all active datasets for a database component and drops its connections:

```
Database1.Connected := False;
```

Note: *Close* disconnects from a database server even if *KeepConnection* is *True*.

Closing datasets without disconnecting from a server

Topic groups See also

There may be times when you want to close all datasets without disconnecting from the database server. To close all open datasets without disconnecting from a server, follow these steps:

1. Set the database component's *KeepConnection* property to *True*.
2. Call the database component's *CloseDataSets* method.

Iterating through a database component's datasets

Topic groups See also

A database component provides two properties that enable an application to iterate through all the datasets associated with the component: *DataSets* and *DataSetCount*.

DataSets is an indexed array of all active datasets (*TTable*, *TQuery*, and *TStoredProc*) for a database component. An active dataset is one that is currently open. *DataSetCount* is a read-only integer value specifying the number of currently active datasets.

You can use *DataSets* with *DataSetCount* to cycle through all currently active datasets in code. For example, the following code cycles through all active datasets to set the *CachedUpdates* property for each dataset of type *TTable* to *True*:

```
var
  I: Integer;
begin
  for I := 0 to DataSetCount - 1 do
    if DataSets[I] is TTable then
      DataSets[I].CachedUpdates := True;
end;
```

Understanding database and session component interactions

Topic groups See also

In general, session component properties, such as *KeepConnection*, provide global, default behaviors that apply to all temporary database components created as needed at runtime.

Session methods apply somewhat differently. *TSession* methods affect all database components, regardless of database component status. For example, the session method *DropConnections*closes all datasets belonging to a session's database components, and then drops all database connections, even if the *KeepConnection* property for individual database components is *True*.

Database component methods apply only to the datasets associated with a given database component. For example, suppose the database component *Database1* is associated with the default session. *Database1.CloseDataSets()* closes only those datasets associated with *Database1*. Open datasets belonging to other database components within the default session remain open.

Using database components in data modules

Topic groups See also

You can safely place database components in data modules. If you put a data module that contains a database component into the Object Repository, however, and you want other users to be able to inherit from it, you must set the *HandleShared* property of the database component to *True* to prevent global name space conflicts.

Executing SQL statements from a TDatabase component

Topic groups See also

Simple SQL statements can be executed directly from a *TDatabase* component using its *Execute* method. The *Execute* method is designed primarily for executing data definition language (DDL) SQL statements. These are statements that do not return result sets and only operate on a database's metadata. The *Execute* method can, however, also be used to execute data manipulation language (DML) SQL statements that return result sets and operate on the data stored in the tables of a database.

Use the *Execute* method to execute one SQL statement at a time. It is not possible to execute multiple SQL statements with a single call to *Execute*, such as with SQL scripting utilities. To execute more than one statement: replace the contents of the SQL parameter with a new SQL statement and call the *Execute* method again, repeating these two steps as many times as there are statements to execute. It may be necessary or appropriate to make other changes as well, such as changing parameter values between statement executions, depending on the actual situation.

Using the *Execute* method to execute SQL statements is described in more detail in:

- Executing SQL statements without result sets
- Executing SQL statements with result sets
- Executing parameterized SQL statements

Executing SQL statements without result sets

Topic groups See also

Data definition language (DDL) SQL statements and some data manipulation language (DML) statements do not produce a result set. These statements are executed, affect metadata (for DDL statements) or data (DML statements), and then cease all interaction with the database. Examples of DDL statements include: CREATE INDEX, ALTER TABLE, and DROP DOMAIN. Examples of DML statements that perform an action on data but do not return a result set include: INSERT, DELETE, and UPDATE.

To execute these statements that do not produce a result set, make a call to the *TDatabase.Execute* method. The SQL statement to be executed is represented by a **String** value (either variable or literal) and used as the SQL parameter of the *Execute* method. If no parameters are used in the SQL statement, pass a **nil** value for the *Params* parameter of *Execute*. For information on using parameters with the *Execute* method, see Executing parameterized SQL statements. As these statements do not return a result set, pass a **nil** value in the *Cursor* parameter of *Execute*.

In the example below, a CREATE TABLE statement (DDL) without any parameters is executed with the *Execute* method.

```
procedure TDataForm.CreateTableButtonClick(Sender: TObject);
var
  SQLstmt: String;
begin
  Database1.Connected := True;
  SQLstmt := 'CREATE TABLE NewCusts ' +
    '( ' +
    '  CustNo INTEGER, ' +
    '  Company CHAR(40), ' +
    '  State CHAR(2), ' +
    '  PRIMARY KEY (CustNo) ' +
    ')';
  Database1.Execute(SQLstmt, nil, False, nil);
end;
```

In the following example, an INSERT statement (DDL) without any parameters and not returning a result set is executed.

```
procedure TDataForm.InsertRecordButtonClick(Sender: TObject);
var
  SQLstmt: String;
begin
  Database1.Connected := True;
  SQLstmt := 'INSERT INTO Customer ' +
    '(Custno, Company, State) ' +
    'VALUES (9999, "John Doe Rentals", "CA")';
  Database1.Execute(SQLstmt, nil, False, nil);
end;
```

Executing SQL statements with result sets

Topic groups See also

Only data manipulation language (DML) SQL statements return result sets. Even then, only a DML query that uses the SELECT statement produces a result set. (The SELECT statement might be against a base table or a stored procedure that returns a result set.)

To execute these statements that produce a result set, make a call to the *TDatabase.Execute* method and provide a dataset component for the result set. The SQL statement to be executed is represented by a **String** value (either variable or literal) and used as the SQL parameter of the *Execute* method. If no parameters are used in the SQL statement, pass a **nil** value for the *Params* parameter of *Execute*. For information on using parameters with the *Execute* method, see Executing parameterized SQL statements.

To make the result set produced by the SQL statement available after the call to the *Execute* method, provide an already-existing dataset component (such as a *TTable* component). Provide a variable of type hDBICur (a handle to a BDE cursor). (To make use of this data type, the unit containing the call to the *Execute* method must include the BDE wrapper unit “BDE” in its Uses section.) Pass this hDBICur variable (dereferenced) in the *Cursor* parameter of the *Execute* method. After *Execute* is called, assign the handle of the BDE cursor to the *Handle* property of the dataset component. (The dataset component must be typecast as *TDBDataSet* to make this assignment because the *Handle* property is read-only in all classes related to datasets except *TDBDataSet*.)

The example routine below executes a SELECT statement using the *Execute* method. The result set produced by this action is made available through a *TTable* component named *Table1*.

```
procedure TDataForm.SELECT_NoParamsButtonClick(Sender: TObject);
var
  SQLstmt: String;
  Cursor: hDBICur;
begin
  Database1.Connected := True;
  SQLstmt := 'SELECT Company, State ' +
    'FROM Customer ' +
    'ORDER BY State, Company';
  Database1.Execute(SQLstmt, nil, False, @Cursor);
  Table1.Close;
  (Table1 as TDBDataSet).Handle := Cursor;
end;
```

If the dataset component was used previously to represent a different SQL result set, call its *Close* method to properly deactivate that data cursor before assigning the new one to the dataset component.

The result set returned through the *Cursor* parameter of the *Execute* method is always read-only.

Executing parameterized SQL statements

Topic groups See also

Some SQL statements include parameters through which data values are passed.

To execute these statements that include parameters, make a call to the *TDatabase.Execute* method and pass an object of type *TParams* in the *Params* parameter. The SQL statement to be executed is represented by a **String** value (either variable or literal) and used as the SQL parameter of the *Execute* method. If a statement does not return a result set, pass a **nil** value in the *Cursor* parameter. If the statement does return a result set, see Executing SQL statements with result sets.

To represent the parameters, use an object of type *TParams*. Pass this *TParams* object as the *Params* parameter of the *Execute* method. Each parameter of the SQL statement is represented in the *TParams* object by a separate *TParam* object. Use the *TParams.CreateParam* method to add one *TParam* object to the *TParams* object for each parameter in the SQL statement. Use properties and methods of *TParam* to set the value of the *TParam* object to give the SQL parameter a value when the *Execute* method is called.

In the example below, an SQL statement using parameters is executed. The SQL statement includes a single parameter named *StateParam*. A *TParams* object (called *stmtParams*) is created within the routine and the *TParams.CreateParam* method is called to add a single *TParam* object to *stmtParams*. After the *TParam* object has been created, it is assigned a value of “CA”. Then the *Execute* method is called using the *stmtParams* object for the *Params* parameter.

```
procedure TDataForm.SELECT_WithParamsButtonClick(Sender: TObject);
var
  SQLstmt: String;
  stmtParams: TParams;
  Cursor: hDBICur;
begin
  stmtParams := TParams.Create;
  try
    Database1.Connected := True;
    stmtParams.CreateParam(ftString, 'StateParam', ptInput);
    stmtParams.ParamByName('StateParam').AsString := 'CA';
    SQLstmt := 'SELECT Company, State '+
      'FROM "Customer.db" ' +
      'WHERE (State = :StateParam) ' +
      'ORDER BY State, Company';
    Database1.Execute(SQLstmt, stmtParams, False, @Cursor);
    Table1.Close;
    TDBDataSet(Table1).Handle := Cursor;
  finally
    stmtParams.Free;
  end;
end;
```

Successfully executing a parameterized SQL statement using the *Execute* method depends on these requirements being fulfilled:

1. Parameters must appear in the SQL statement (these being tokens prefixed with colons).
2. A *TParams* object must be created to contain *TParam* objects.
3. One *TParam* object must be created in the *TParams* object for each parameter in the SQL statement.
4. The *TParams* object must be used for the *Params* parameter of the *Execute* method.

If a parameter token is in the SQL statement but no *TParam* object is created to represent it, the parameter in the SQL statement cannot be given a value and the SQL statement may cause an error when executed (depends on the particular database back-end used). If a *TParam* object is provided but there is no corresponding parameter token in the SQL statement, an exception is raised when the application attempts to use the *TParam*.

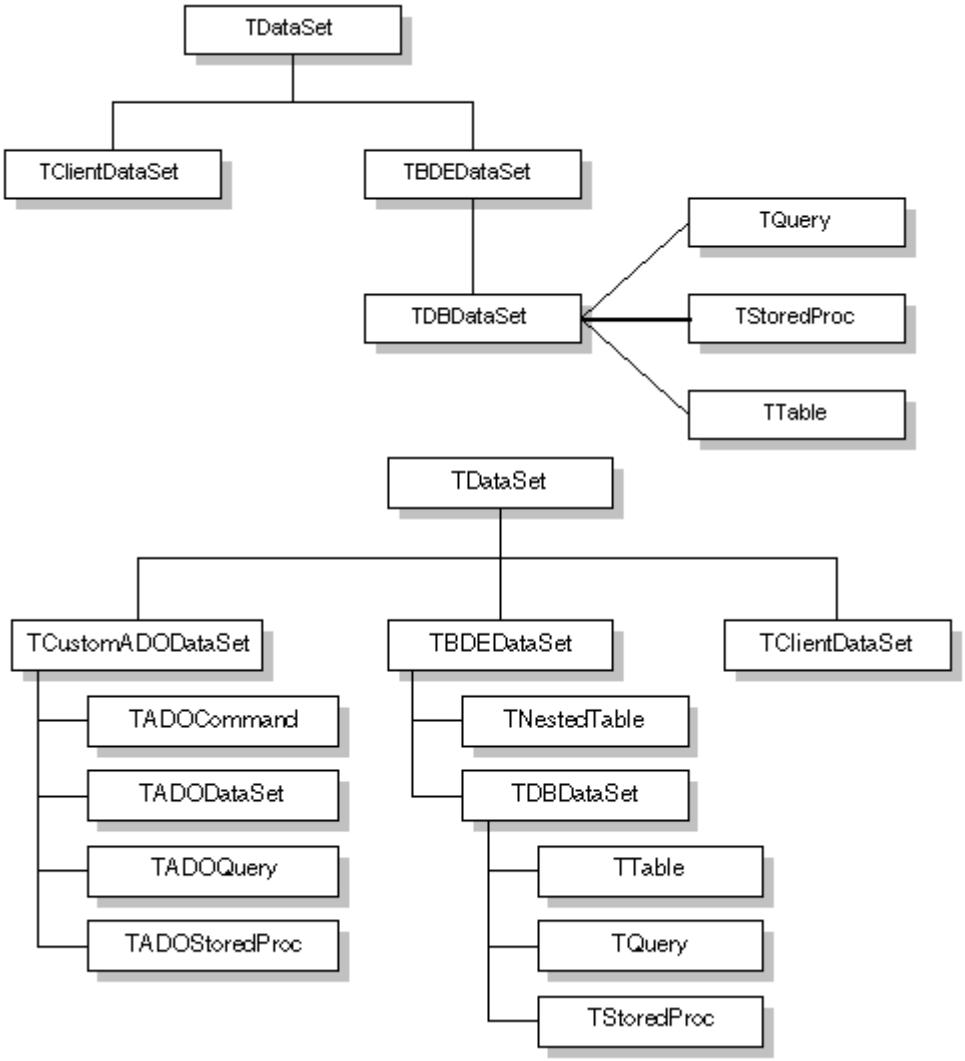
Understanding datasets

Topic groups See also

In Delphi, the fundamental unit for accessing data is the dataset family of objects. Your application uses datasets for all database access. Generally, a dataset object represents a specific table belonging to a database, or it represents a query or stored procedure that accesses a database.

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

The following figure illustrates the hierarchical relationship of all the dataset components:



What is TDataSet?

Topic groups See also

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

Because *TDataSet* contains **abstract** methods, you cannot use it directly in an application without generating a runtime error. Instead, you either create instances of *TDataSet*'s descendants, such as TTable, TQuery, TStoredProc, and TClientDataSet, and use them in your application, or you derive your own dataset object from *TDataSet* or its descendants and write implementations for all its **abstract** methods.

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

The following topics are discussed in this section:

- Types of datasets
- Opening and closing datasets
- Determining and setting dataset states
- Navigating datasets
- Searching datasets
- Displaying and editing a subset of data using filters
- Modifying data
- Using dataset events
- Using BDE-enabled datasets

Types of datasets

Topic groups

To understand the concepts common to all dataset objects, and to prepare for developing your own custom dataset objects that do not rely on either the Borland Database Engine (BDE) or ActiveX Data Objects (ADO), read this section.

To develop traditional, two-tier client/server database applications using the Borland Database Engine (BDE), see "Overview of BDE-enablement." That section introduces TBDEDataSet and TDBDataSet, and focuses on the shared features of TQuery, TStoredProc, and TTable, the dataset components used most commonly in all database applications.

With some versions of Delphi, you can develop multi-tier database applications using distributed datasets. To learn about working with client datasets in multi-tiered applications, see "Creating multi-tiered applications." That section discusses how to use TClientDataSet and connect the client to an application server.

Opening and closing datasets

Topic groups See also

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

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

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

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

```
CustQuery.Open;
```

You can close a dataset in two ways,

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

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

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

```
CustTable.Close;
```

You may need to close a dataset when you want to change certain of its properties, such as *TableName* on a *TTable* component. At runtime, you may also want to close a dataset for other reasons specific to your application.

Determining and setting dataset states

Topic groups See also

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

Value	State	Meaning
dsInactive	Inactive	DataSet closed. Its data is unavailable.
dsBrowse	Browse	DataSet open. Its data can be viewed, but not changed. This is the default state of an open dataset.
dsEdit	Edit	DataSet open. The current row can be modified.
dsInsert	Insert	DataSet open. A new row is inserted or appended.
dsSetKey	SetKey	*TTable* and *TClientDataSet* only. DataSet open. Enables setting of ranges and key values used for ranges and *GotoKey* operations.
dsCalcFields	CalcFields	DataSet open. Indicates that an *OnCalcFields* event is under way. Prevents changes to fields that are not calculated.
dsCurValue	CurValue	Internal use only.
dsNewValue	NewValue	Internal use only.
dsOldValue	OldValue	Internal use only.
dsFilter	Filter	DataSet open. Indicates that a filter operation is under way. A restricted set of data can be viewed, and no data can be changed.

When an application opens a dataset, it appears by default in *dsBrowse* mode. The state of a dataset changes as an application processes data. An open dataset changes from one state to another based on either the

- code in your application, or
- built-in behavior of data-related components.

To put a dataset into *dsBrowse*, *dsEdit*, *dsInsert*, or *dsSetKey* states, call the method corresponding to the name of the state. For example, the following code puts *CustTable* into *dsInsert* state, accepts user input for a new record, and writes the new record to the database:

```
CustTable.Insert; { Your application explicitly sets dataset state to Insert }
AddressPromptDialog.ShowModal;
if AddressPromptDialog.ModalResult := mrOK then
  CustTable.Post; { Delphi sets dataset state to Browse on successful completion }
else
  CustTable.Cancel; {Delphi sets dataset state to Browse on cancel }
```

This example also illustrates that the state of a dataset automatically changes to *dsBrowse* when

- The Post method successfully writes a record to the database. (If *Post* fails, the dataset state remains unchanged.)
- The Cancel method is called.

Some states cannot be set directly. For example, to put a dataset into *dsInactive* state, set its *Active* property to *False*, or call the *Close* method for the dataset. The following statements are equivalent:

```
CustTable.Active := False;
CustTable.Close;
```

The remaining states (*dsCalcFields*, *dsCurValue, dsNewValue*, *dsOldValue*, and *dsFilter*) cannot be set by your application. Instead, the state of the dataset changes automatically to these values as necessary. For example, *dsCalcFields* is set when a dataset's *OnCalcFields* event is called. When the *OnCalcFields* event finishes, the dataset is restored to its previous state.

Note: Whenever a dataset's state changes, the *OnStateChange* event is called for any data source

components associated with the dataset. For more information about data source components and *OnStateChange*, see "Using TDataSource events".

The following sections provide overviews of each state, how and when states are set, how states relate to one another, and where to go for related information, if applicable.

The following topics are discussed in this section:

- Inactivating a dataset
- Browsing a dataset
- Enabling dataset editing
- Enabling insertion of new records
- Enabling index-based searches and ranges on tables
- Calculating fields
- Filtering records
- Updating records

Inactivating a dataset

Topic groups See also

A dataset is inactive when it is closed. You cannot access records in a closed dataset. At design time, a dataset is closed until you set its *Active* property to *True*. At runtime, a dataset is initially closed until an application opens it by calling the *Open* method, or by setting the *Active* property to *True*.

When you open an inactive dataset, its state automatically changes to the *dsBrowse* state.

To make a dataset inactive, call its *Close* method. You can write *BeforeClose* and *AfterClose* event handlers that respond to the *Close* method for a dataset. For example, if a dataset is in *dsEdit* or *dsInsert* modes when an application calls *Close*, you should prompt the user to post pending changes or cancel them before closing the dataset. The following code illustrates such a handler:

```
procedure CustTable.VerifyBeforeClose(DataSet: TDataSet)
begin
  if (CustTable.State = dsEdit) or (CustTable.State = dsInsert) then
  begin
    if MessageDlg('Post changes before closing?', mtConfirmation, mbYesNo, 0) =
mrYes then
      CustTable.Post;
    else
      CustTable.Cancel;
  end;
end;
```

To associate a procedure with the *BeforeClose* event for a dataset at design time:

1. Select the table in the data module (or form).
2. Click the Events page in the Object Inspector.
3. Enter the name of the procedure for the *BeforeClose* event (or choose it from the drop-down list).

Browsing a dataset

Topic groups See also

When an application opens a dataset, the dataset automatically enters *dsBrowse* state. Browsing enables you to view records in a dataset, but you cannot edit records or insert new records. You mainly use *dsBrowse* to scroll from record to record in a dataset. For more information about scrolling from record to record, see "Navigating datasets."

From *dsBrowse* all other dataset states can be set. For example, calling the *Insert* or *Append* methods for a dataset changes its state from *dsBrowse* to *dsInsert* (note that other factors and dataset properties, such as *CanModify*, may prevent this change). Calling *SetKey* to search for records puts a dataset in *dsSetKey* mode. For more information about inserting and appending records in a dataset, see "Modifying data."

Two methods associated with all datasets can return a dataset to *dsBrowse* state. *Cancel* ends the current edit, insert, or search task, and always returns a dataset to *dsBrowse* state. *Post* attempts to write changes to the database, and if successful, also returns a dataset to *dsBrowse* state. If *Post* fails, the current state remains unchanged.

Enabling dataset editing

Topic groups See also

A dataset must be in *dsEdit* mode before an application can modify records. In your code you can use the *Edit* method to put a dataset into *dsEdit* mode if the read-only *CanModify* property for the dataset is *True*. *CanModify* is *True* if the database underlying a dataset permits read and write privileges.

On forms in your application, some data-aware controls can automatically put a dataset into *dsEdit* state if:

- The control's *ReadOnly* property is *False* (the default),
- The *AutoEdit* property of the data source for the control is *True*, and
- *CanModify* is *True* for the dataset.

Important: For *TTable* components, if the *ReadOnly* property is *True*, *CanModify* is *False*, preventing editing of records. Similarly, for *TQuery* components, if the *RequestLive* property is *False*, *CanModify* is *False*.

Note: Even if a dataset is in *dsEdit* state, editing records may not succeed for SQL-based databases if your application user does not have proper SQL access privileges.

You can return a dataset from *dsEdit* state to *dsBrowse* state in code by calling the *Cancel*, *Post*, or *Delete* methods. *Cancel* discards edits to the current field or record. *Post* attempts to write a modified record to the dataset, and if it succeeds, returns the dataset to *dsBrowse*. If *Post* cannot write changes, the dataset remains in *dsEdit* state. *Delete* attempts to remove the current record from the dataset, and if it succeeds, returns the dataset to *dsBrowse* state. If *Delete* fails, the dataset remains in *dsEdit* state.

Data-aware controls for which editing is enabled automatically call *Post* when a user executes any action that changes the current record (such as moving to a different record in a grid) or that causes the control to lose focus (such as moving to a different control on the form).

For a complete discussion of editing fields and records in a dataset, see "Modifying data."

Enabling insertion of new records

Topic groups See also

A dataset must be in *dsInsert* mode before an application can add new records. In your code you can use the *Insert* or *Append* methods to put a dataset into *dsInsert* mode if the read-only *CanModify* property for the dataset is *True*. *CanModify* is *True* if the database underlying a dataset permits read and write privileges.

On forms in your application, the data-aware grid and navigator controls can put a dataset into *dsInsert* state if

- The control's *ReadOnly* property is *False* (the default),
- The *AutoEdit* property of the data source for the control is *True*, and
- *CanModify* is *True* for the dataset.

Important: For *TTable* components, if the *ReadOnly* property is *True*, *CanModify* is *False*, preventing editing of records. Similarly, for *TQuery* components, if the *RequestLive* property is *False*, *CanModify* is *False*.

Note: Even if a dataset is in *dsInsert* state, inserting records may not succeed for SQL-based databases if your application user does not have proper SQL access privileges.

You can return a dataset from *dsInsert* state to *dsBrowse* state in code by calling the *Cancel*, *Post*, or*Delete* methods. *Delete* and *Cancel* discard the new record. *Post* attempts to write the new record to the dataset, and if it succeeds, returns the dataset to *dsBrowse*. If *Post* cannot write the record, the dataset remains in *dsInsert* state.

Data-aware controls for which inserting is enabled automatically call *Post* when a user executes any action that changes the current record (such as moving to a different record in a grid).

For more discussion of inserting and appending records in a dataset, see "Modifying data."

Enabling index-based searches and ranges on tables

Topic groups See also

You can search against any dataset using the *Locate* and *Lookup* methods of *TDataSet*. *TTable* components, however, provide an additional family of *GotoKey* and *FindKey* methods that enable you to search for records based on an index for the table. To use these methods on table components, the component must be in *dsSetKey* mode. *dsSetKey* mode applies only to *TTable* components. You put a dataset into *dsSetKey* mode with the *SetKey* method at runtime. The *GotoKey*, *GotoNearest*, *FindKey*, and *FindNearest* methods, which carry out searches, returns the dataset to *dsBrowse* state upon completion of the search. For more information about searching a table based on its index, see "Searching for records based on indexed fields.".

You can temporarily view and edit a subset of data for any dataset by using filters. For more information about filters, see "Displaying and editing a subset of data using filters." *TTable* components also support an additional way to access a subset of available records, called ranges. To create and apply a range to a table, a table must be in *dsSetKey* mode. For more information about using ranges, see "Working with a subset of data.".

Calculating fields

Topic groups See also

Delphi puts a dataset into *dsCalcFields* mode whenever an application calls the dataset's *OnCalcFields* event handler. This state prevents modifications or additions to the records in a dataset except for the calculated fields the handler is designed to modify. The reason all other modifications are prevented is because *OnCalcFields* uses the values in other fields to derive values for calculated fields. Changes to those other fields might otherwise invalidate the values assigned to calculated fields.

When the *OnCalcFields* handler finishes, the dataset is returned to *dsBrowse* state.

For more information about creating calculated fields and *OnCalcFields* event handlers, see "Using OnCalcFields."

Filtering records

Topic groups See also

Delphi puts a dataset into *dsFilter* mode whenever an application calls the dataset's *OnFilterRecord* event handler. This state prevents modifications or additions to the records in a dataset during the filtering process so that the filter request is not invalidated. For more information about filtering, see "Displaying and editing a subset of data using filters."

When the *OnFilterRecord* handler finishes, the dataset is returned to *dsBrowse* state.

Updating records

Topic groups See also

When performing cached update operations, Delphi may put the dataset into *dsNewValue*, *dsOldValue*, or *dsCurValue* states temporarily. These states indicate that the corresponding properties of a field component (*NewValue*, *OldValue*, and *CurValue*, respectively) are being accessed, usually in an *OnUpdateError* event handler. Your applications cannot see or set these states.

Navigating datasets

Topic groups See also

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

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

Method	Description
First	Moves the cursor to the first row in a dataset.
Last	Moves the cursor to the last row in a dataset.
Next	Moves the cursor to the next row in a dataset.
Prior	Moves the cursor to the previous row in a dataset.
MoveBy	Moves the cursor a specified number of rows forward or back in a dataset.

The data-aware, visual component *TDBNavigator* encapsulates these methods as buttons that users can click to move among records at runtime. For more information about the navigator component, see "Using data controls."

In addition to these methods, the following table describes two Boolean properties of datasets that provide useful information when iterating through the records in a dataset.

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

The following topics are discussed in this section:

- Using the First and Last methods
- Using the Next and Prior methods
- Using the MoveBy method
- Using the Eof and Bof properties
- Marking and returning to records

Using the First and Last methods

Topic groups See also

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

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

```
CustTable.First;
```

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

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

```
CustTable.Last;
```

Note: While there may be programmatic reasons to move to the first or last rows in a dataset without user intervention, you should enable your users to navigate from record to record using the *TDBNavigator* component. The navigator component contains buttons that when active and visible enables a user to move to the first and last rows of an active dataset. The *OnClick* events for these buttons call the *First* and *Last* methods of the dataset. For more information about making effective use of the navigator component, see "Using data controls."

Using the Next and Prior methods

Topic groups See also

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

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

```
CustTable.Next;
```

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

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

```
CustTable.Prior;
```

Using the MoveBy method

Topic groups See also

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

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

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

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

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

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

Using the Eof and Bof properties

Topic groups See also

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

The following topics are discussed in this section:

- Eof
- Bof

Eof

Topic groups See also

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

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

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

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

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

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

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

Bof

Topic groups See also

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

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

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

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

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

Marking and returning to records

Topic groups See also

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

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

TDataSet implements **virtual** bookmark methods. While these methods ensure that any dataset object derived from *TDataSet* returns a value if a bookmark method is called, the return values are merely defaults that do not keep track of the current location. Descendants of *TDataSet*, such as *TBDEDataSet*, reimplement the bookmark methods to return meaningful values as described in the following list:

- *BookmarkValid*, for determining if a specified bookmark is in use.
- *CompareBookmarks*, to test two bookmarks to see if they are the same.
- *GetBookmark*, to allocate a bookmark for your current position in the dataset.
- *GotoBookmark*, to return to a bookmark previously created by *GetBookmark*
- *FreeBookmark*, to free a bookmark previously allocated by *GetBookmark*.

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

Before calling *GotoBookmark* to move to a specific record, you can call *BookmarkValid* to determine if the bookmark points to a record. *BookmarkValid* returns *True* if a specified bookmark points to a record. In *TDataSet*, *BookmarkValid* is a virtual method that always returns *False*, indicating that the bookmark is not valid. *TDataSet* descendants reimplement this method to provide a meaningful return value.

You can also call *CompareBookmarks* to see if a bookmark you want to move to is different from another (or the current) bookmark. *TDataSet.CompareBookmarks* always returns 0, indicating that the bookmarks are identical. *TDataSet* descendants reimplement this method to provide a meaningful return value.

When passed a bookmark, *GotoBookmark* moves the cursor for the dataset to the location specified in the bookmark. *TDataSet.GotoBookmark* calls an internal pure virtual method which generates a runtime error if called. *TDataSet* descendants reimplement this method to provide a meaningful return value.

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

The following code illustrates one use of bookmarking:

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

```
      Tbl.FreeBookmark(Bookmark); {Deallocate memory for the bookmark }
    end;
  end;
```

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

Searching datasets

Topic groups See also

You can search any dataset for specific records using the generic search methods *Locate* and *Lookup*. These methods enable you to search on any type of columns in any dataset.

These two methods are discussed in the following topics:

- Using Locate
- Using Lookup

Using Locate

Topic groups See also

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

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

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

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

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

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

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

Using Lookup

Topic groups See also

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

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

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

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

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

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

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

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

Displaying and editing a subset of data using filters

Topic groups See also

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

A filter specifies conditions a record must meet to be displayed. Filter conditions can be stipulated in a dataset's *Filter* property or coded into its *OnFilterRecord* event handler. Filter conditions are based on the values in any specified number of fields in a dataset whether or not those fields are indexed. For example, to view only those records for companies based in California, a simple filter might require that records contain a value in the State field of "CA".

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

The following topics are discussed in this section:

- Enabling and disabling filtering
- Navigating records in a filtered dataset

Enabling and disabling filtering

Topic groups See also

Enabling filters on a dataset is a three-step process:

1. Create a filter.
2. Set filter options for string-based filter tests, if necessary.
3. Set the *Filtered* property to *True*.

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

Creating filters

Topic groups See also

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

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

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

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

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

Setting the Filter property

Topic groups See also

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

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

You can also supply a value for *Filter* based on the text entered in a control. For example, the following statement assigns the text in an edit box to *Filter*:

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

You can, of course, create a string based on both hard-coded text and data entered by a user in a control:

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

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

You can also compare field values to literals, and to constants using the following comparison and logical operators:

Operator	Meaning
<	Less than
>	Greater than
>=	Greater than or equal to
<=	Less than or equal to
=	Equal to
<>	Not equal to
AND	Tests two statements are both *True*
NOT	Tests that the following statement is not *True*
OR	Tests that at least one of two statements is *True*

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

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

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

Writing an OnFilterRecord event handler

Topic groups See also

A filter for a dataset is an event handler that responds to *OnFilterRecord* events generated by the dataset for each record it retrieves. At the heart of every filter handler is a test that determines if a record should be included in those that are visible to the application.

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

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

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

You can code any number of filter event handlers and switch among them at runtime. To switch to a different filter event handler at runtime, assign the new event handler to the dataset’s *OnFilterRecord* property.

For example, the following statements switch to an *OnFilterRecord* event handler called *NewYorkFilter*:

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

Setting filter options

Topic groups See also

The *FilterOptions* property enables you to specify whether or not a filter that compares string-based fields accepts records based on partial comparisons and whether or not string comparisons are case-sensitive. *FilterOptions* is a set property that can be an empty set (the default), or that can contain either or both of the following values:

Value	Meaning
foCaseInsensitive	Ignore case when comparing strings.
foPartialCompare	Disable partial string matching (i.e., do not match strings ending with an asterisk (*)).

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

```
FilterOptions := [foCaseInsensitive];
Filter := '''State'' = ''CA''';
```

Navigating records in a filtered dataset

Topic groups See also

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

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

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

```
DataSet1.FindFirst;
```

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

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

Note: If filtering is disabled and you do not set the *Filter* property or create an *OnFilterRecord* event handler, these methods do the same thing as *First*(), *Last*(), *Next*(), and *Prior*().

All navigational filter methods position the cursor on a matching record (if one if found) make that record the current one, and return *True*. If a matching record is not found, the cursor position is unchanged, and these methods return *False*. You can check the status of the *Found* property to wrap these calls, and only take action when *Found* is *True*. For example, if the cursor is already on the last matching record in the dataset, and you call *FindNext*, the method returns *False*, and the current record is unchanged.

Modifying data

Topic groups See also

You can use the following dataset methods to insert, update, and delete data:

Method	Description
Edit	Puts the dataset into *dsEdit* state if it is not already in *dsEdit* or *dsInsert* states.
Append	Posts any pending data, moves current record to the end of the dataset, and puts the dataset in *dsInsert* state.
Insert	Posts any pending data, and puts the dataset in *dsInsert* state.
Post	Attempts to post the new or altered record to the database. If successful, the dataset is put in *dsBrowse* state; if unsuccessful, the dataset remains in its current state.
Cancel	Cancels the current operation and puts the dataset in *dsBrowse* state.
Delete	Deletes the current record and puts the dataset in *dsBrowse* state.

The following topics discuss these methods in greater detail:

- Editing records
- Adding new records
- Deleting records
- Posting data to the database
- Canceling changes
- Modifying entire records

Editing records

Topic groups See also

A dataset must be in *dsEdit* mode before an application can modify records. In your code you can use the *Edit* method to put a dataset into *dsEdit* mode if the read-only *CanModify* property for the dataset is *True*. *CanModify* is *True* if the table(s) underlying a dataset permits read and write privileges.

On forms in your application, some data-aware controls can automatically put a dataset into *dsEdit* state if

- The control's *ReadOnly* property is *False* (the default),
- The *AutoEdit* property of the data source for the control is *True*, and
- *CanModify* is *True* for the dataset.

Important: For *TTable* components with the *ReadOnly* property set to *True* and *TQuery* components with the *RequestLive* property set to *False*, *CanModify* is *False*, preventing editing of records.

Note: Even if a dataset is in *dsEdit* state, editing records may not succeed for SQL-based databases if your application's user does not have proper SQL access privileges.

Once a dataset is in *dsEdit* mode, a user can modify the field values for the current record that appears in any data-aware controls on a form. Data-aware controls for which editing is enabled automatically call *Post* when a user executes any action that changes the current record (such as moving to a different record in a grid).

If you provide a navigator component on your forms, users can cancel edits by clicking the navigator's Cancel button. Canceling edits returns a dataset to *dsBrowse* state.

In code, you must write or cancel edits by calling the appropriate methods. You write changes by calling *Post*. You cancel them by calling *Cancel*. In code, *Edit* and *Post* are often used together. For example,

```
with CustTable do
begin
  Edit;
  FieldValues['CustNo'] := 1234;
  Post;
end;
```

In the previous example, the first line of code places the dataset in *dsEdit* mode. The next line of code assigns the number 1234 to the *CustNo* field of the current record. Finally, the last line writes, or posts, the modified record to the database.

Note: If the *CachedUpdates* property for a dataset is *True*, posted modifications are written to a temporary buffer. To write cached edits to the database, call the *ApplyUpdates* method for the dataset. For more information about cached updates, see "Working with cached updates."

Adding new records

Topic groups See also

A dataset must be in *dsInsert* mode before an application can add new records. In code, you can use the *Insert* or *Append* methods to put a dataset into *dsInsert* mode if the read-only *CanModify* property for the dataset is *True*. *CanModify* is *True* if the database underlying a dataset permits read and write privileges.

On forms in your application, the data-aware grid and navigator controls can put a dataset into *dsInsert* state if

- The control's *ReadOnly* property is *False* (the default), and
- *CanModify* is *True* for the dataset.

Once a dataset is in *dsInsert* mode, a user or application can enter values into the fields associated with the new record. Except for the grid and navigational controls, there is no visible difference to a user between *Insert* and *Append*. On a call to *Insert*, an empty row appears in a grid above what was the current record. On a call to *Append*, the grid is scrolled to the last record in the dataset, an empty row appears at the bottom of the grid, and the *Next* and *Last* buttons are dimmed on any navigator component associated with the dataset.

Data-aware controls for which inserting is enabled automatically call *Post* when a user executes any action that changes which record is current (such as moving to a different record in a grid). Otherwise you must call *Post* in your code.

Post writes the new record to the database, or, if cached updates are enabled, *Post* writes the record to a buffer. To write cached inserts and appends to the database, call the *ApplyUpdates* method for the dataset.

The following topics are discussed in this section:

- Inserting records
- Appending records

Inserting records

Topic groups See also

Insert opens a new, empty record before the current record, and makes the empty record the current record so that field values for the record can be entered either by a user or by your application code.

When an application calls *Post* (or *ApplyUpdates* when cached updating is enabled), a newly inserted record is written to a database in one of three ways:

- For indexed Paradox and dBASE tables, the record is inserted into the dataset in a position based on its index.
- For unindexed tables, the record is inserted into the dataset at its current position.
- For SQL databases, the physical location of the insertion is implementation-specific. If the table is indexed, the index is updated with the new record information.

Appending records

Topic groups See also

Append opens a new, empty record at the end of the dataset, and makes the empty record the current one so that field values for the record can be entered either by a user or by your application code.

When an application calls *Post* (or *ApplyUpdates* when cached updating is enabled), a newly appended record is written to a database in one of three ways:

- For indexed Paradox and dBASE tables, the record is inserted into the dataset in a position based on its index.
- For unindexed tables, the record is added to the end of the dataset.
- For SQL databases, the physical location of the append is implementation-specific. If the table is indexed, the index is updated with the new record information.

Deleting records

Topic groups See also

A dataset must be active before an application can delete records. *Delete* deletes the current record from a dataset and puts the dataset in *dsBrowse* mode. The record that followed the deleted record becomes the current record. If cached updates are enabled for a dataset, a deleted record is only removed from the temporary cache buffer until you call *ApplyUpdates*.

If you provide a navigator component on your forms, users can delete the current record by clicking the navigator's Delete button. In code, you must call *Delete* explicitly to remove the current record.

Posting data to the database

Topic groups See also

The *Post* method is central to a Delphi application's interaction with a database table. *Post* writes changes to the current record to the database, but it behaves differently depending on a dataset's state.

- In *dsEdit* state, *Post* writes a modified record to the database (or buffer if cached updates is enabled).
- In *dsInsert* state, *Post* writes a new record to the database (or buffer if cached updates is enabled).
- In *dsSetKey* state, *Post* returns the dataset to *dsBrowse* state.

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

Note: The *Close* method does not call *Post* implicitly. Use the *BeforeClose* event to post any pending edits explicitly.

Canceling changes

Topic groups See also

An application can undo changes made to the current record at any time, if it has not yet directly or indirectly called *Post*. For example, if a dataset is in *dsEdit* mode, and a user has changed the data in one or more fields, the application can return the record back to its original values by calling the *Cancel* method for the dataset. A call to *Cancel* always returns a dataset to *dsBrowse* state.

On forms, you can allow users to cancel edit, insert, or append operations by including the Cancel button on a navigator component associated with the dataset, or you can provide code for your own Cancel button on the form.

Modifying entire records

Topic groups See also

On forms, all data-aware controls except for grids and the navigator provide access to a single field in a record.

In code, however, you can use the following methods that work with entire record structures provided that the structure of the database tables underlying the dataset is stable and does not change. The following table summarizes the methods available for working with entire records rather than individual fields in those records:

Method	Description
AppendRecord([array of values])	Appends a record with the specified column values at the end of a table; analogous to *Append*. Performs an implicit *Post*.
InsertRecord([array of values])	Inserts the specified values as a record before the current cursor position of a table; analogous to *Insert*. Performs an implicit *Post*.
SetFields([array of values])	Sets the values of the corresponding fields; analogous to assigning values to *TField*s. Application must perform an explicit *Post*.

These method take an array of *Tavern* values as an argument, where each value corresponds to a column in the underlying dataset. Use the ARRAYOFCONST macro to create these arrays. The values can be literals, variables, or NULL. If the number of values in an argument is less than the number of columns in a dataset, then the remaining values are assumed to be NULL.

For unindexed datasets, *AppendRecord* adds a record to the end of the dataset and *InsertRecord* inserts a record after the current cursor position. For indexed tables, both methods place the record in the correct position in the table, based on the index. In both cases, the methods move the cursor to the record's position.

SetFields assigns the values specified in the array of parameters to fields in the dataset. To use *SetFields*, an application must first call *Edit* to put the dataset in *dsEdit* mode. To apply the changes to the current record, it must perform a *Post*.

If you use *SetFields* to modify some, but not all fields in an existing record, you can pass NULL values for fields you do not want to change. If you do not supply enough values for all fields in a record, SetFields assigns NULL values to them. NULL values overwrite any existing values already in those fields.

For example, suppose a database has a COUNTRY table with columns for Name, Capital, Continent, Area, and Population. If a *TTable* component called *CountryTable* were linked to the COUNTRY table, the following statement would insert a record into the COUNTRY table:

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

This statement does not specify values for Area and Population, so NULL values are inserted for them. The table is indexed on Name, so the statement would insert the record based on the alphabetic collation of "Japan".

To update the record, an application could use the following code:

```
with CountryTable do
begin
  if Locate('Name', 'Japan', loCaseInsensitive) then;
  begin
    Edit;
    SetFields(nil, nil, nil, 344567, 164700000);
    Post;
  end;
end;
```

This code assigns values to the Area and Population fields and then posts them to the database. The three NULL pointers act as place holders for the first three columns to preserve their current contents.

Warning: When using NULL pointers with *SetFields* to leave some field values untouched, be sure to cast the NULL to a void *. If you use NULL as a parameter without the cast, you will set the field to a blank value.

Using dataset events

Topic groups See also

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

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 about events for the *TDataSet* component, see TDataSet.

The following topics are discussed in this section:

- Aborting a method
- Using OnCalcFields

Aborting a method

Topic groups See also

To abort a method such as an *Open* or *Insert*, call the *Abort* procedure in any of the *Before* event handlers (*BeforeOpen*, *BeforeInsert*, and so on). For example, the following code requests a user to confirm a delete operation or else it aborts the call to *Delete*:

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

Using OnCalcFields

Topic groups See also

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

- A dataset is opened.
- Focus moves from one visual component to another, or from one column to another in a data-aware grid control and the current record has been modified.
- A record is retrieved from the database.

OnCalcFields is always called whenever a value in a non-calculated field changes, regardless of the setting of *AutoCalcFields*.

Caution: *OnCalcFields* is called frequently, so the code you write for it should be kept short. Also, if *AutoCalcFields* is *True*, *OnCalcFields* should not perform any actions that modify the dataset (or 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* is called again, leading to another *Post*, and so on.

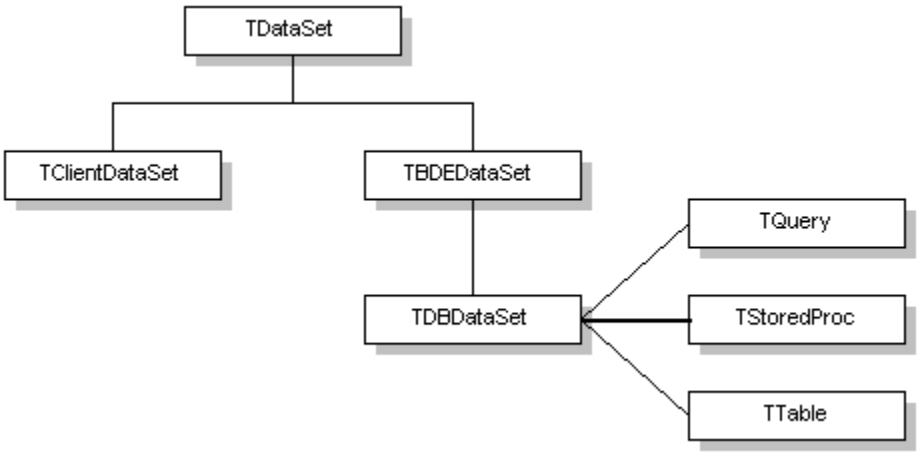
If *AutoCalcFields* is *False*, then *OnCalcFields* is not called when individual fields within a single record are modified.

When *OnCalcFields* executes, a dataset is in *dsCalcFields* mode, so you cannot set the values of any fields other than calculated fields. After *OnCalcFields* is completed, the dataset returns to *dsBrowse* state.

Using BDE-enabled datasets

Topic groups See also

BDE-enabled datasets provide functionality to the dataset components that use the Borland Database Engine (BDE) to access data. Support for BDE-enablement occurs in TBDEDataSet, which is a direct descendant of TDataSet. Additional database and session control features occur in TDBDataSet, which is a direct descendant of *TBDEDataSet*.



This section introduces the dataset features provided by *TBDEDataSet* and *TDBDataSet* in Overview of BDE-enablement. It assumes you are already familiar with *TDataSet discussed earlier in this section*. For a general understanding of dataset components descended from *TDataSet*, see "Understanding datasets."

Note: Although you need to understand the functionality provided by *TBDEDataSet* and *TDBDataSet*, unless you develop your own custom BDE-enabled datasets, you never use *TBDEDataSet* and *TDBDataSet* directly in your applications. Instead, you use the direct descendants of *TDBDataSet*: *TQuery*, *TStoredProc*, and *TTable*. For specific information about using *TStoredProc*, see "Working with stored procedures." For specific information about using *TQuery*, see "Working with queries." For specific information about *TTable*, see "Working with tables."

Overview of BDE-enablement

Topic groups See also

The TBDEDataSet component implements the abstract methods of *TDataSet* that control record navigation, indexing, and bookmarking. It also reimplements many of *TDataSet*'s virtual methods and events to take advantage of the BDE. The BDE-specific implementations of *TDataSet*'s features do not depart from the general description about using these features with *TDataSet*, so for more information about them, see "Understanding datasets."

In addition to BDE-specific features common to all datasets, *TBDEDataSet* introduces new properties, events, and methods for handling BLOBs, cached updates, and communicating with a remote database server. TDBDataSet introduces a method and properties for handling database connections and associating a dataset with a BDE session. The following sections describe these features.

- Handling database and session connections
- Using cached updates
- Caching BLOBs

Handling database and session connections

Topic groups See also

The *TDBDataSet* component introduces the following properties and function for working with database and session connections:

Function or property	Purpose
CheckOpen function	Determines if a database is open. Returns *True* if the connection is active, *False* otherwise.
Database	Identifies the database component with which the dataset is associated.
DBHandle	Specifies the BDE database handle for the database component specified in the *Database* property. Used only when making some direct BDE API calls.
DBLocale	Specifies the BDE locale information for the database component specified in the *Database* property. Used only when making some direct BDE API calls.
DBSession	Specifies the BDE session handle for the session component specified by the *SessionName* property. Used only when making some direct BDE API calls.
DatabaseName	Specifies the BDE alias or database component name for the database used by this dataset. If the dataset is a Paradox or dBASE table, *DatabaseName* can be a full path specification for the database's directory location.
SessionName	Specifies the session with which this dataset component is associated. If you use both database and session components with a dataset, the setting for *SessionName* should be the same as the database component's *SessionName* property.

The following topics provide more detail on these properties:

- Using the DatabaseName and SessionName properties
- Working with BDE handle properties

Using the DatabaseName and SessionName properties

Topic groups See also

Of the *TDBDataSet* database and session properties, the most commonly used are *DatabaseName* and *SessionName*. If you work with databases on a remote database server, such as Sybase, Oracle, or InterBase, your application usually maintains that connection through a *TDatabase* component. You should set the *DatabaseName* property of each dataset to match the name of the database component that establishes the database connection used by the dataset. If you do not use database components, *DatabaseName* should be set to a BDE alias (or, optionally, a full path specification for dBASE and Paradox).

SessionName indicates the BDE session with which to associate a dataset. If you do not use explicit session components in your application, you do not have to provide a value for this property. It is supplied for you. If your application provides more than one session, you can set a dataset's *SessionName* property to match the *SessionName* property of the appropriate session component in your application. If your application uses both multiple session components and one or more database components, the *SessionName* property for a dataset must match the *SessionName* property for the database component with which the dataset is associated.

For more information about handling database connections with *TDatabase*, see "Connecting to databases." For more information about managing sessions with *TSession* and *TSessionList*, see "Managing database sessions."

Working with BDE handle properties

Topic groups See also

Unless you bypass the built-in functionality of dataset components and make direct API calls to the BDE, you do not need to use the *DBHandle*, *DBLocale*, and *DBSession* properties. These properties are read-only properties that are automatically assigned to a dataset when it is connected to a database server through the BDE. These properties are provided as a resource for application developers who need to make direct API calls to BDE functions, some of which take handle parameters. For more information about the BDE API, see the online help file, BDE32.HLP.

Working with cached updates

Topic groups See also

Cached updates enable you to retrieve data from a database, cache and edit it locally, and then apply the cached updates to the database as a unit. When cached updates are enabled, updates to a dataset (such as posting changes or deleting records) are stored in an internal cache instead of being written directly to the dataset's underlying table. When changes are complete, your application calls a method that writes the cached changes to the database and clears the cache.

The recommended approach when caching updates is to use a client dataset rather than a BDE-enabled dataset. However, *TBDEDataSet* provides an alternate approach, with built-in methods for handling cached updates. The following table lists the relevant properties, events, and methods for cached updating:

Property, event, or method	Purpose
CachedUpdates property	Determines whether or not cached updates are in effect for the dataset. If *True*, cached updating is enabled. If *False*, cached updating is disabled.
UpdateObject property	Indicates the name of the *TUpdateSQL* component used to update datasets based on queries.
UpdatesPending property	Indicates whether or not the local cache contains updated records that need to be applied to the database. *True* indicates there are records to update. *False* indicates the cache is empty.
UpdateRecordTypes property	Indicates the kind of updated records to make visible to the application during the application of cached updates.
UpdateStatus method	Indicates if a record is unchanged, modified, inserted, or deleted.
OnUpdateError event	A developer-created procedure that handles update errors on a record-by-record basis.
OnUpdateRecord event	A developer-created procedure that processes updates on a record-by-record basis.
ApplyUpdates method	Applies records in the local cache to the database.
CancelUpdates method	Removes all pending updates from the local cache without applying them to the database.
CommitUpdates method	Clears the update cache following successful application of updates.
FetchAll method	Copies all database records to the local cache for editing and updating.
RevertRecord method	Undoes updates to the current record if updates are not yet applied on the server side.

Using cached updates and coordinating them with other applications that access data in a multi-user environment is an advanced topic that is fully covered in Working with cached updates. For information about using a client dataset instead, see Creating and using a client dataset.

Caching BLOBs

Topic groups See also

TBDEDataSet provides the *CacheBlobs* property to control whether BLOB fields are cached locally by the BDE when an application reads BLOB records. By default, *CacheBlobs* is *True*, meaning that the BDE caches a local copy of BLOB fields. Caching BLOBs improves application performance by enabling the BDE to store local copies of BLOBs instead of fetching them repeatedly from the database server as a user scrolls through records.

In applications and environments where BLOBs are frequently updated or replaced, and a fresh view of BLOB data is more important than application performance, you can set *CacheBlobs* to *False* to ensure that your application always sees the latest version of a BLOB field.

Working with field components

Topic groups See also

This section describes the properties, events, and methods common to the TField object and its descendants. Descendants of *TField* represent individual database columns in datasets. This section also describes how to use descendant field components to control the display and editing of data in your applications.

You never use a *TField* component directly in your applications. By default, when you first place a dataset in your application and open it, Delphi automatically assigns a dynamic, data-type-specific descendant of *TField* to represent each column in the database table(s). At design time, you can override dynamic field defaults by invoking the Fields editor to create persistent fields that replace these defaults.

The following table lists each descendant field component, its standard purpose, and, where appropriate, the range of values it can represent:

Component name	Purpose
TADTField	An ADT (Abstract Data Type) field.
TAggregateField	A maintained aggregate in a client dataset.
TArrayField	An array field.
TAutoIncField	Whole number with a range of -2,147,483,648 to 2,147,483,647. Used in Paradox for fields whose values are automatically incremented.
TBCDField	Real number with a fixed number of decimal places, accurate to 18 digits. Range depends on the number of decimal places.
TBooleanField	*True* or *False* values.
TBlobField	Binary data: Theoretical maximum limit: 2GB.
TBytesField	Binary data: Theoretical maximum limit: 2GB.
TCurrencyField	Real numbers with a range of 5.0 * 10-324 to 1.7 * 10308. Used in Paradox for fields with two decimals of precision.
TDataSetField	Nested data set value.
TDateField	Date value.
TDateTimeField	Date and time value.
TFloatField	Real numbers with a range of 5.0 * 10-324 to 1.7 * 10308.
TBytesField	Binary data: Maximum number of bytes: 255.
TIntegerField	Whole number with a range of -2,147,483,648 to 2,147,483,647.
TLargeintField	Whole number with a range of -263 to 2 63.
TMemoField	Text data: Theoretical maximum limit: 2GB.
TNumericField	Real numbers with a range of 3.4 * 10-4932 to 1.1 * 104932
TReferenceField	A pointer to an object relational database object.
TSmallintField	Whole number with a range of -32,768 to 32,768.
TStringField	String data: Maximum size in bytes: 8192, including a null termination character.
TTimeField	Time value.
TVarBytesField	Binary data: Maximum number of bytes: 255.
TWordField	Whole numbers with a range of 0 to 65,535.

This section discusses the properties and methods all field components inherit from *TField*. In many cases, *TField* declares or implements standard functionality that the descendant objects override. When several descendant objects share overridden functionality, that functionality is also described in this section and noted for your convenience. For complete information about individual field components, see the online VCL Reference.

The following topics are discussed in this section:

- Understanding field components
- Creating persistent fields
- Arranging persistent fields
- Defining new persistent fields
- Setting persistent field properties and events
- Working with field component methods at runtime
- Displaying, converting, and accessing field values
- Checking a field’s current value
- Setting a default value for a field
- Working with constraints

Understanding field components

Topic groups See also

Like all Delphi data access components, field components are nonvisual. Field components are also not directly visible at design time. Instead they are associated with a dataset component and provide data-aware components such as *TDBEdit* and *TDBGrid* access to database columns through that dataset.

Generally speaking, a single field component represents the characteristics of a single column in a database field, such as its data type and size. It also represents the field's display characteristics, such as alignment, display format, and edit format. Finally, as you scroll from record to record within a dataset, a field component also enables you to view and change the value for that field in the current record. For example, a *TFloatField* component has four properties that directly affect the appearance of its data:

Property	Purpose
Alignment	Specifies whether data is displayed left-aligned, centered, or right-aligned.
DisplayWidth	Specifies the number of digits to display in a control at one time.
DisplayFormat	Specifies data formatting for display (such as how many decimal places to show).
EditFormat	Specifies how to display a value during editing.

Field components have many properties in common with one another (such as *DisplayWidth* and *Alignment*), and they have properties specific to their data types (such as *Precision* for *TFloatField*). Each of these properties affect how data appears to an application's users on a form. Some properties, such as *Precision*, can also affect what data values the user can enter in a control when modifying or entering data.

All field components for a dataset are either *dynamic* (automatically generated for you based on the underlying structure of database tables), or *persistent* (generated based on specific field names and properties you set in the Fields editor). Dynamic and persistent fields have different strengths and are appropriate for different types of applications. The following sections describe dynamic and persistent fields in more detail and offer advice on choosing between them.

- Dynamic field components
- Persistent field components

Dynamic field components

Topic groups See also

Dynamically generated field components are the default. In fact, all field components for any dataset start out as dynamic fields the first time you place a dataset on a data module, associate the dataset with a database, and open it. A field component is *dynamic* if it is created automatically based on the underlying physical characteristics of the columns in one or more database tables accessed by a dataset. Delphi generates one field component for each column in the underlying tables or query. The exact *TField* descendant created for each column in an underlying database table is determined by field type information received from the Borland Database Engine (BDE) or (in multi-tiered applications) a provider component.

A field component's type determines its properties and how data associated with that field is displayed in data-aware controls on a form. Dynamic fields are temporary. They exist only as long as a dataset is open.

Each time you reopen a dataset that uses dynamic fields, Delphi rebuilds a completely new set of dynamic field components for it based on the current structure of the database tables underlying the dataset. If the columns in those database tables are changed, then the next time you open a dataset that uses dynamic field components, the automatically generated field components are also changed to match.

Use dynamic fields in applications that must be flexible about data display and editing. For example, to create a database exploration tool like the SQL Explorer, you must use dynamic fields because every database table has different numbers and types of columns. You might also want to use dynamic fields in applications where user interaction with data mostly takes place inside grid components and you know that the database tables used by the application change frequently.

To use dynamic fields in an application:

1. Place datasets and data sources in a data module.
2. Associate the datasets with database tables and queries, and associate the data sources with the datasets.
3. Place data-aware controls in the application's forms, add the data module to each uses clause for each form's unit, and associate each data-aware control with a data source in the module. In addition, associate a field with each data-aware control that requires one.
4. Open the datasets.

Aside from ease of use, dynamic fields can be limiting. Without writing code, you cannot change the display and editing defaults for dynamic fields, you cannot safely change the order in which dynamic fields are displayed, and you cannot prevent access to any fields in the dataset. You cannot create additional fields for the dataset, such as calculated fields or lookup fields, and you cannot override a dynamic field's default data type. To gain control and flexibility over fields in your database applications, you need to invoke the Fields editor to create persistent field components for your datasets.

Persistent field components

Topic groups See also

By default, dataset fields are dynamic. Their properties and availability are automatically set and cannot be changed in any way. To gain control over a field's properties and events so that you can set or change the field's visibility or display characteristics at design time or runtime, create new fields based on existing fields in a dataset, or validate data entry, you must create persistent fields for the dataset.

At design time, you can—and should—use the Fields editor to create persistent lists of the field components used by the datasets in your application. Persistent field component lists are stored in your application, and do not change even if the structure of a database underlying a dataset is changed.

Creating persistent field components offers the following advantages. You can:

- Restrict the fields in your dataset to a subset of the columns available in the underlying database.
- Add field components to the list of persistent components.
- Remove field components from the list of persistent components to prevent your application from accessing particular columns in an underlying database.
- Define new fields—usually to replace existing fields—based on columns in the table or query underlying a dataset.
- Define calculated fields that compute their values based on other fields in the dataset.
- Define lookup fields that compute their values based on fields in other datasets.
- Modify field component display and edit properties.

A persistent field is one that Delphi generates based on field names and properties you specify in the Fields editor. Once you create persistent fields with the Fields editor, you can also create event handlers for them that respond to changes in data values and that validate data entries.

Note: When you create persistent fields for a dataset, only those fields you select are available to your application at design time and runtime. At design time, you can always choose Add Fields from the Fields editor to add or remove persistent fields for a dataset.

All fields used by a single dataset are either persistent or dynamic. You cannot mix field types in a single dataset. If you create persistent fields for a dataset, and then want to revert to dynamic fields, you must remove all persistent fields from the dataset. For more information about dynamic fields, see "Dynamic field components."

Note: One of the primary uses of persistent fields is to gain control over the appearance and display of data. You can also control data appearance in other ways. For example, you can use the Data Dictionary to assign field attributes to a field component. You can also control the appearance of columns in data-aware grids. For more information about the Data Dictionary, see "Creating attribute sets for field components." To learn about controlling column appearance in grids, see Creating a customized grid.

Creating persistent fields

Topic groups See also

Persistent field components created with the Fields editor provide efficient, readable, and type-safe programmatic access to underlying data. Using persistent field components guarantees that each time your application runs, it always uses and displays the same columns, in the same order even if the physical structure of the underlying database has changed. Data-aware components and program code that rely on specific fields always work as expected. If a column on which a persistent field component is based is deleted or changed, Delphi generates an exception rather than running the application against a nonexistent column or mismatched data.

To create persistent fields for a dataset:

1. Place a dataset in a data module.
2. Set the *DatabaseName* property for the dataset.
3. Set the *TableName* property (for a *TTable*), or the SQL property (for a *TQuery*).
4. Double-click the dataset component in the data module to invoke the Fields editor. The Fields editor contains a title bar, navigator buttons, and a list box.

   The title bar of the Fields editor displays both the name of the data module or form containing the dataset, and the name of the dataset itself. For example, if you open the *Customers* dataset in the *CustomerData* data module, the title bar displays 'CustomerData.Customers,' or as much of the name as fits.

   Below the title bar is a set of navigation buttons that enable you to scroll one-by-one through the records in an active dataset at design time, and to jump to the first or last record. The navigation buttons are dimmed if the dataset is not active or if the dataset is empty.

   The list box displays the names of persistent field components for the dataset. The first time you invoke the Fields editor for a new dataset, the list is empty because the field components for the dataset are dynamic, not persistent. If you invoke the Fields editor for a dataset that already has persistent field components, you see the field component names in the list box.
5. Choose Add Fields from the Fields editor context menu.
6. Select the fields to make persistent in the Add Fields dialog box. By default, all fields are selected when the dialog box opens. Any fields you select become persistent fields.

The Add Fields dialog box closes, and the fields you selected appear in the Fields editor list box. Fields in the Fields editor list box are persistent. If the dataset is active, note, too, that the Next and Last navigation buttons above the list box are enabled.

From now on, each time you open the dataset, Delphi no longer creates dynamic field components for every column in the underlying database. Instead it only creates persistent components for the fields you specified.

Each time you open the dataset, Delphi verifies that each non-calculated persistent field exists or can be created from data in the database. If it cannot, it raises an exception warning you that the field is not valid, and does not open the dataset.

Arranging persistent fields

Topic groups See also

The order in which persistent field components are listed in the Fields editor list box is the default order in which the fields appear in a data-aware grid component. You can change field order by dragging and dropping fields in the list box.

To change the order of fields:

1. Select the fields. You can select and order one or more fields at a time.
2. Drag the fields to a new location.

If you select a noncontiguous set of fields and drag them to a new location, they are inserted as a contiguous block. Within the block, the order of fields does not change.

Alternatively, you can select the field, and use *Ctrl+Up* and *Ctrl+Dn* to change an individual field's order in the list.

Defining new persistent fields

Topic groups See also

Besides making existing dataset fields into persistent fields, you can also create special persistent fields as additions to or replacements of the other persistent fields in a dataset. The following table lists the types of additional fields you can create:

Field kind	Purpose
Data	Replaces an existing field (for example to change its data type, based on columns in the table or query underlying a dataset.)
Calculated	Displays values calculated at runtime by a dataset's *OnCalcFields* event handler.
InternalCalc	Displays values calculated at runtime by a client dataset and stored with its data.
Lookup	Retrieve values from a specified dataset at runtime based on search criteria you specify.
Aggregate	Displays a summary value of the data in a set of records.

These types of persistent fields are only for display purposes. The data they contain at runtime are not retained either because they already exist elsewhere in your database, or because they are temporary. The physical structure of the table and data underlying the dataset is not changed in any way.

To create a new persistent field component, invoke the context menu for the Fields editor and choose New field. The New Field dialog box appears.

The New Field dialog box contains three group boxes: Field properties, Field type, and Lookup definition.

The Field type radio group enables you to specify the type of new field component to create. The default type is Data. If you choose Lookup, the Dataset and Source Fields edit boxes in the Lookup definition group box are enabled. You can also create Calculated fields, and if you're working with a *TClientDataSet*component, you can also create InternalCalc fields.

The Field properties group box enables you to enter general field component information. Enter the component's field name in the Name edit box. The name you enter here corresponds to the field component's *FieldName* property. Delphi uses this name to build a component name in the Component edit box. The name that appears in the Component edit box corresponds to the field component's *Name* property and is only provided for informational purposes (*Name* contains the identifier by which you refer to the field component in your source code). Delphi discards anything you enter directly in the Component edit box.

The Type combo box in the Field properties group enables you to specify the field component's data type. You must supply a data type for any new field component you create. For example, to display floating-point currency values in a field, select *Currency* from the drop-down list. The Size edit box enables you to specify the maximum number of characters that can be displayed or entered in a string-based field, or the size of *Bytes* and *VarBytes* fields. For all other data types, Size is meaningless.

The Lookup definition group box is only used to create lookup fields.

The following topics are discussed in this section:

- Defining a data field
- Defining a calculated field
- Programming a calculated field
- Defining a lookup field
- Deleting persistent field components

Defining a data field

See also

A data field replaces an existing field in a dataset. For example, for programmatic reasons you might want to replace a *TSmallIntField* with a *TIntegerField*. Because you cannot change a field's data type directly, you must define a new field to replace it.

Important: Even though you define a new field to replace an existing field, the field you define must derive its data values from an existing column in a table underlying a dataset.

To create a replacement data field for a field in a table underlying a dataset, follow these steps:

1. Remove the field from the list of persistent fields assigned for the dataset, and then choose New Field from the context menu.
2. In the New Field dialog box, enter the name of an existing field in the database table in the Name edit box. Do not enter a new field name. You are actually specifying the name of the field from which your new field will derive its data.
3. Choose a new data type for the field from the Type combo box. The data type you choose should be different from the data type of the field you are replacing. You cannot replace a string field of one size with a string field of another size. Note that while the data type should be different, it must be compatible with the actual data type of the field in the underlying table.
4. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.
5. Select Data in the Field type radio group if it is not already selected.
6. Choose OK. The New Field dialog box closes, the newly defined data field replaces the existing field you specified in Step 1, and the component declaration in the data module or form's **type** declaration is updated.

To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the Object Inspector. For more information about editing field component properties and events, see "Setting persistent field properties and events."

Defining a calculated field

See also

A calculated field displays values calculated at runtime by a dataset's *OnCalcFields* event handler. For example, you might create a string field that displays concatenated values from other fields.

To create a calculated field in the New Field dialog box:

1. Enter a name for the calculated field in the Name edit box. Do not enter the name of an existing field.
2. Choose a data type for the field from the Type combo box.
3. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.
4. Select Calculated in the Field type radio group.
5. Choose OK. The newly defined calculated field is automatically added to the end of the list of persistent fields in the Field editor list box, and the component declaration is automatically added to the form's **type** declaration in the source code.
6. Place code that calculates values for the field in the *OnCalcFields*event handler for the dataset. For more information about writing code to calculate field values, see "Programming a calculated field."

Note: To edit the properties or events associated with the field component, select the component name in the Field editor list box, then edit its properties or events with the Object Inspector. For more information about editing field component properties and events, see "Setting persistent field properties and events."

If you are working with a client dataset or query component, you can also create an InternalCalc field. You create and program an internally calculated field just like you do a calculated field. For a client dataset, the significant difference between these types of calculated fields is that the values calculated for an InternalCalc field are stored and retrieved as part of the client dataset's data. To create an InternalCalc field, select the InternalCalc radio button in the Field type group.

Programming a calculated field

See also

After you define a calculated field, you must write code to calculate its value. Otherwise, it always has a null value. Code for a calculated field is placed in the *OnCalcFields* event for its dataset.

To program a value for a calculated field:

1. Select the dataset component from the Object Inspector drop-down list.
2. Choose the Object Inspector Events page.
3. Double-click the *OnCalcFields* property to bring up or create a *CalcFields* procedure for the dataset component.
4. Write the code that sets the values and other properties of the calculated field as desired.

For example, suppose you have created a *CityStateZip* calculated field for the *Customers* table on the *CustomerData* data module. *CityStateZip* should display a company's city, state, and zip code on a single line in a data-aware control.

To add code to the *CalcFields* procedure for the *Customers* table, select the *Customers* table from the Object Inspector drop-down list, switch to the Events page, and double-click the *OnCalcFields* property.

The *TCustomerData.CustomersCalcFields* procedure appears in the unit's source code window. Add the following code to the procedure to calculate the field:

```
CustomersCityStateZip.Value := CustomersCity.Value + ', ' + CustomersState.Value
  + ' ' + CustomersZip.Value;
```

Defining a lookup field

See also

A lookup field is a read-only field that displays values at runtime based on search criteria you specify. In its simplest form, a lookup field is passed the name of an existing field to search on, a field value to search for, and a different field in a lookup dataset whose value it should display.

For example, consider a mail-order application that enables an operator to use a lookup field to determine automatically the city and state that correspond to the zip code a customer provides. The column to search on might be called *ZipTable.Zip*, the value to search for is the customer's zip code as entered in *Order.CustZip*, and the values to return would be those for the *ZipTable.City* and *ZipTable.State* columns of the record where the value of *ZipTable.Zip* matches the current value in the *Order.CustZip* field.

To create a lookup field in the New Field dialog box:

1. Enter a name for the lookup field in the Name edit box. Do not enter the name of an existing field.
2. Choose a data type for the field from the Type combo box.
3. Enter the size of the field in the Size edit box, if appropriate. Size is only relevant for fields of type *TStringField*, *TBytesField*, and *TVarBytesField*.
4. Select Lookup in the Field type radio group. Selecting Lookup enables the Dataset and Key Fields combo boxes.
5. Choose from the Dataset combo box drop-down list the dataset in which to look up field values. The lookup dataset must be different from the dataset for the field component itself, or a circular reference exception is raised at runtime. Specifying a lookup dataset enables the Lookup Keys and Result Field combo boxes.
6. Choose from the Key Fields drop-down list a field in the current dataset for which to match values. To match more than one field, enter field names directly instead of choosing from the drop-down list. Separate multiple field names with semicolons. If you are using more than one field, you must use persistent field components.
7. Choose from the Lookup Keys drop-down list a field in the lookup dataset to match against the Source Fields field you specified in step 6. If you specified more than one key field, you must specify the same number of lookup keys. To specify more than one field, enter field names directly, separating multiple field names with semicolons.
8. Choose from the Result Field drop-down list a field in the lookup dataset to return as the value of the lookup field you are creating.

When you design and run your application, lookup field values are determined before calculated field values are calculated. You can base calculated fields on lookup fields, but you cannot base lookup fields on calculated fields. You can use the *LookupCache* property to hone this behavior. *LookupCache* determines whether the values of a lookup field are cached in memory when a dataset is first opened, or looked up dynamically every time the current record in the dataset changes.

Set *LookupCache* to *True* to cache the values of a lookup field when the *LookupDataSet* is unlikely to change and the number of distinct lookup values is small. Caching lookup values can speed performance, because the lookup values for every set of *LookupKeyFields* values are preloaded when the *DataSet* is opened. When the current record in the *DataSet* changes, the field object can locate its *Value* in the cache, rather than accessing the *LookupDataSet*. This performance improvement is especially dramatic if the *LookupDataSet* is on a network where access is slow.

Tip: You can use a lookup cache to provide lookup values programmatically rather than from a secondary dataset. Create a *TLookupList* object at runtime, and use its *Add* method to fill it with lookup values. Set the *LookupList* property of the lookup field to this *TLookupList* object and set its *LookupCache* property to *True*. If the other lookup properties of the field are not set, the field will use the supplied lookup list without overwriting it with values from a lookup dataset.

If every record of *DataSet* has different values for *KeyFields*, the overhead of locating values in the cache can be greater than any performance benefit provided by the cache. The overhead of locating values in the cache increases with the number of distinct values that can be taken by *KeyFields*.

If *LookupDataSet* is volatile, caching lookup values can lead to inaccurate results. Call

tRefreshLookupLis to update the values in the lookup cache. *RefreshLookupList* regenerates the *LookupList* property, which contains the value of the *LookupResultField* for every set of *LookupKeyFields* values.

When setting *LookupCache* at runtime, call *RefreshLookupList* to initialize the cache.

Defining an aggregate field

See also

An aggregate field displays values from a maintained aggregate in a client dataset. An aggregate is a calculation that summarizes the data in a set of records.

To create an aggregate field in the New Field dialog box:

1. Enter a name for the aggregate field in the Name edit box. Do not enter the name of an existing field.
2. Choose aggregate data type for the field from the Type combo box.
3. Select Aggregate in the Field type radio group.
4. Choose OK. The newly defined aggregate field is automatically added to the client dataset's *Aggregates* is automatically updated to include the appropriate aggregate specification, and the component declaration is automatically added to the form's **type** declaration in the source code.
5. Place the calculation for the aggregate in the *ExprText* property of the newly created aggregate field. For more information about defining an aggregate, see "Specifying aggregates".

Once a persistent *TAggregateField* is created, a TDBText control can be bound to the aggregate field. The TDBText control will then display the value of the aggregate field that is relevant to the current record of the underlying client data set.

Deleting persistent field components

See also

Deleting a persistent field component is useful for accessing a subset of available columns in a table, and for defining your own persistent fields to replace a column in a table. To remove one or more persistent field components for a dataset:

1. Select the field(s) to remove in the Fields editor list box.
2. Press *Del*.

Note: You can also delete selected fields by invoking the context menu and choosing Delete.

Fields you remove are no longer available to the dataset and cannot be displayed by data-aware controls. You can always re-create persistent field components that you delete by accident, but any changes previously made to its properties or events is lost. For more information, see "Creating persistent fields."

Note: If you remove all persistent field components for a dataset, the dataset reverts to using dynamic field components for every column in the underlying database table.

Setting persistent field properties and events

See also

You can set properties and customize events for persistent field components at design time. Properties control the way a field is displayed by a data-aware component, for example, whether it can appear in a *TDBGrid*, or whether its value can be modified. Events control what happens when data in a field is fetched, changed, set, or validated.

To set the properties of a field component or write customized event handlers for it, select the component in the Fields editor, or select it from the component list in the Object Inspector.

The following topics are discussed in this section:

- Setting display and edit properties at design time
- Setting field component properties at runtime
- Creating attribute sets for field components
- Associating attribute sets with field components
- Removing attribute associations
- Controlling and masking user input
- Using default formatting for numeric, date, and time fields
- Handling events

Setting display and edit properties at design time

See also

To edit the display properties of a selected field component, switch to the Properties page on the Object Inspector window. The following table summarizes display properties that can be edited.

Property	Purpose
Alignment	Left justifies, right justifies, or centers a field contents within a data-aware component.
ConstraintErrorMessage	Specifies the text to display when edits clash with a constraint condition.
CustomConstraint	Specifies a local constraint to apply to data during editing.
Currency	Numeric fields only. *True*: displays monetary values.
	False (default): does not display monetary values.
DisplayFormat	Specifies the format of data displayed in a data-aware component.
DisplayLabel	Specifies the column name for a field in a data-aware grid component.
DisplayWidth	Specifies the width, in characters, of a grid column that display this field.
EditFormat	Specifies the edit format of data in a data-aware component.
EditMask	Limits data-entry in an editable field to specified types and ranges of characters, and specifies any special, non-editable characters that appear within the field (hyphens, parentheses, and so on).
FieldKind	Specifies the type of field to create.
FieldName	Specifies the actual name of a column in the table from which the field derives its value and data type.
HasConstraints	Indicates whether or not there are constraint conditions imposed on a field.
ImportedConstraint	Specifies an SQL constraint imported from the Data Dictionary or an SQL server.
Index	Specifies the order of the field in a dataset.
LookupDataSet	Specifies the table used to look up field values when *Lookup* is *True*.
LookupKeyFields	Specifies the field(s) in the lookup dataset to match when doing a lookup.
LookupResultField	Specifies the field in the lookup dataset from which to copy values into this field.
MaxValue	Numeric fields only. Specifies the maximum value a user can enter for the field.
MinValue	Numeric fields only. Specifies the minimum value a user can enter for the field.
Name	Specifies the component name of the field component within Delphi.
Origin	Specifies the name of the field as it appears in the underlying database.
Precision	Numeric fields only. Specifies the number of significant digits.
ReadOnly	*True*: Displays field values in data-aware components, but prevents editing.
	False (the default): Permits display and editing of field values.
Size	Specifies the maximum number of characters that can be displayed or entered in a string-based field, or the size, in bytes, of *TBytesField* and *TVarBytesField* fields.

Tag	Long integer bucket available for programmer use in every component as needed.
Transliterate	*True* (default): specifies that translation to and from the respective locales will occur as data is transferred between a dataset and a database. *False*: Locale translation does not occur.
Visible	*True* (the default): Permits display of field in a data-aware grid component. *False*: Prevents display of field in a data-aware grid component. User-defined components can make display decisions based on this property.

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

While the purpose of most properties is straightforward, some properties, such as *Calculated*, require additional programming steps to be useful. Others, such as *DisplayFormat*, *EditFormat*, and *EditMask*, are interrelated; their settings must be coordinated. For more information about using *DisplayFormat*, *EditFormat*, and *EditMask*, see "Controlling and masking user input."

Setting field component properties at run time

See also

You can use and manipulate the properties of field component at runtime. For example, the following code sets the *ReadOnly* property for the *CityStateZip* field in the *Customers* table to *True*:

```
CustomersCityStateZip.ReadOnly := True;
```

And this statement changes field ordering by setting the *Index* property of the *CityStateZip* field in the *Customers* table to 3:

```
CustomersCityStateZip.Index := 3;
```

Creating attribute sets for field components

See also

When several fields in the datasets used by your application share common formatting properties (such as *Alignment*, *DisplayWidth*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, and so on), it is more convenient to set the properties for a single field, then store those properties as an attribute set in the Data Dictionary. Attribute sets stored in the data dictionary can be easily applied to other fields.

To create an attribute set based on a field component in a dataset:

1. Double-click the dataset to invoke the Fields editor.
2. Select the field for which to set properties.
3. Set the desired properties for the field in the Object Inspector.
4. Right-click the Fields editor list box to invoke the context menu.
5. Choose Save Attributes to save the current field's property settings as an attribute set in the Data Dictionary.

The name for the attribute set defaults to the name of the current field. You can specify a different name for the attribute set by choosing Save Attributes As instead of Save Attributes from the context menu.

Note: You can also create attribute sets directly from the SQL Explorer. When you create an attribute set from the data dictionary, it is not applied to any fields, but you can specify two additional attributes: a field type (such as *TFloatField*, *TStringField*, and so on) and a data-aware control (such as *TDBEdit*, *TDBCheckBox*, and so on) that is automatically placed on a form when a field based on the attribute set is dragged to the form. For more information, see the online help for the SQL Explorer.

Associating attribute sets with field components

See also

When several fields in the datasets used by your application share common formatting properties (such as *Alignment*, *DisplayWidth*, *DisplayFormat*, *EditFormat*, *MaxValue*, *MinValue*, and so on), and you have saved those property settings as attribute sets in the Data Dictionary, you can easily apply the attribute sets to fields without having to recreate the settings manually for each field. In addition, if you later change the attribute settings in the Data Dictionary, those changes are automatically applied to every field associated with the set the next time field components are added to the dataset.

To apply an attribute set to a field component:

1. Double-click the dataset to invoke the Fields editor.
2. Select the field for which to apply an attribute set.
3. Invoke the context menu and choose Associate Attributes.
4. Select or enter the attribute set to apply from the Associate Attributes dialog box. If there is an attribute set in the Data Dictionary that has the same name as the current field, that set name appears in the edit box.

Important: If the attribute set in the Data Dictionary is changed at a later date, you must reapply the attribute set to each field component that uses it. You can invoke the Fields editor to multi-select field components within a dataset to which to reapply attributes.

Removing attribute associations

See also

If you change your mind about associating an attribute set with a field, you can remove the association by following these steps:

1. Invoke the Fields editor for the dataset containing the field.
2. Select the field or fields from which to remove the attribute association.
3. Invoke the context menu for the Fields editor and choose Unassociate Attributes.

Important: Unassociating an attribute set does not change any field properties. A field retains the settings it had when the attribute set was applied to it. To change these properties, select the field in the Fields editor and set its properties in the Object Inspector.

Controlling and masking user input

See also

The *EditMask* property provides a way to control the type and range of values a user can enter into a data-aware component associated with *TStringField*, *TDateField*, *TTimeField*, and *TDateTimeField* components. You can use existing masks, or create your own. The easiest way to use and create edit masks is with the Input Mask editor. You can, however, enter masks directly into the *EditMask* field in the Object Inspector.

Note: For *TStringField* components, the *EditMask* property is also its display format.

To invoke the Input Mask editor for a field component:

1. Select the component in the Fields editor or Object Inspector.
2. Click the Properties page in the Object Inspector.
3. Double-click the values column for the EditMask field in the Object Inspector, or click the ellipsis button. The Input Mask editor opens.

The Input Mask edit box enables you to create and edit a mask format. The Sample Masks grid lets you select from predefined masks. If you select a sample mask, the mask format appears in the Input Mask edit box where you can modify it or use it as is. You can test the allowable user input for a mask in the Test Input edit box.

The Masks button enables you to load a custom set of masks—if you have created one—into the Sample Masks grid for easy selection.

Using default formatting for numeric, date, and time fields

See also

Delphi provides built-in display and edit format routines and intelligent default formatting for *TFloatField*, *TCurrencyField*, *TIntegerField*, *TSmallIntField*, *TWordField*, *TDateField*, *TDateTimeField*, and *TTimeField* components. To use these routines, you need do nothing.

Default formatting is performed by the following routines:

Routine	Used by . . .
FormatFloat	*TFloatField*, *TCurrencyField*
FormatDateTime	*TDateField*, *TTimeField*, *TDateTimeField*
FormatCurr	*TCurrencyField*

Only format properties appropriate to the data type of a field component are available for a given component.

Default formatting conventions for date, time, currency, and numeric values are based on the Regional Settings properties in the Control Panel. For example, using the default settings for the United States, a *TFloatField* column with the *Currency* property set to *True* sets the *DisplayFormat* property for the value 1234.56 to $1234.56, while the *EditFormat* is 1234.56.

At design time or runtime, you can edit the *DisplayFormat* and *EditFormat* properties of a field component to override the default display settings for that field. You can also write *OnGetText* and *OnSetText* event handlers to do custom formatting for field components at runtime. For more information about setting field component properties at runtime, see "Setting field component properties at runtime."

Handling events

See also

Like most components, field components have event handlers associated with them. By writing these handlers you can control events that affect data entered in fields through data-aware controls. The following table lists the events associated with field components:

Event	Purpose
OnChange	Called when the value for a field changes.
OnGetText	Called when the value for a field component is retrieved for display or editing.
OnSetText	Called when the value for a field component is set.
OnValidate	Called to validate the value for a field component whenever the value is changed because of an edit or insert operation.

OnGetText and *OnSetText* events are primarily useful to programmers who want to do custom formatting that goes beyond the built-in formatting functions. *OnChange* is useful for performing application-specific tasks associated with data change, such as enabling or disabling menus or visual controls. *OnValidate* is useful when you want to control data-entry validation in your application before returning values to a database server.

To write an event handler for a field component:

1. Select the component.
2. Select the Events page in the Object Inspector.
3. Double-click the Value field for the event handler to display its source code window.
4. Create or edit the handler code.

Working with field component methods at runtime

See also

Field components methods available at runtime enable you to convert field values from one data type to another, and enable you to set focus to the first data-aware control in a form that is associated with a field component.

Controlling the focus of data-aware components associated with a field is important when your application performs record-oriented data validation in a dataset event handler (such as *BeforePost*). Validation may be performed on the fields in a record whether or not its associated data-aware control has focus. Should validation fail for a particular field in the record, you want the data-aware control containing the faulty data to have focus so that the user can enter corrections.

You control focus for a field's data-aware components with a field's *FocusControl* method. *FocusControl* sets focus to the first data-aware control in a form that is associated with a field. An event handler should call a field's *FocusControl* method before validating the field. The following code illustrates how to call the *FocusControl* method for the *Company* field in the *Customers* table:

```
CustomersCompany.FocusControl;
```

The following table lists some other field component methods and their uses. For a complete list and detailed information about using each method, see TField.

Method	Purpose
AssignValue	Sets a field value to a specified value using an automatic conversion function based on the field's type.
Clear	Clears the field and sets its value to NULL.
GetData	Retrieves unformatted data from the field.
IsValidChar	Determines if a character entered by a user in a data-aware control to set a value is allowed for this field.
SetData	Assigns unformatted data to this field.

Displaying, converting, and accessing field values

See also

Data-aware controls such as *TDBEdit* and *TDBGrid* automatically display the values associated with field components. If editing is enabled for the dataset and the controls, data-aware controls can also send new and changed values to the database. In general, the built-in properties and methods of data-aware controls enable them to connect to datasets, display values, and make updates without requiring extra programming on your part. Use them whenever possible in your database applications. For more information about data-aware control, see "Using data controls"

Standard controls can also display and edit database values associated with field components. Using standard controls, however, may require additional programming on your part.

The following topics are discussed in this section:

- Displaying field component values in standard controls
- Converting field values
- Accessing field values with the default dataset property
- Accessing field values with a dataset's Fields property
- Accessing field values with a dataset's FieldByName method

Displaying field component values in standard controls

See also

An application can access the value of a database column through the *Value* property of a field component. For example, the following statement assigns the value of the *CustomersCompany* field to the text in a *TEdit* control:

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

This method works well for string values, but may require additional programming to handle conversions for other data types. Fortunately, field components have built-in functions for handling conversions.

Note: You can also use variants to access and set field values. Variants are a new and flexible data type. For more information about using variants to access and set field values, see "Accessing field values with the default dataset property."

Converting field values

See also

Conversion functions attempt to convert one data type to another. For example, the *AsString* function converts numeric and Boolean values to string representations. The following table lists field component conversion functions, and which functions are recommended for field components by field-component type:

Note that the *AsVariant* method is recommended to translate among all data types. When in doubt, use *AsVariant*.

In some cases, conversions are not always possible. For example, *AsDateTime* can be used to convert a string to a date, time, or datetime format only if the string value is in a recognizable datetime format. A failed conversion attempt raises an exception.

In some other cases, conversion is possible, but the results of the conversion are not always intuitive. For example, what does it mean to convert a *TDateTimeField* value into a float format? *AsFloat* converts the date portion of the field to the number of days since 12/31/1899, and it converts the time portion of the field to a fraction of 24 hours. The following table lists permissible conversions that produce special results:

Conversion	Result
String to Boolean	Converts "*True*," "*False*," "Yes," and "No" to Boolean. Other values raise exceptions.
Float to Integer	Rounds float value to nearest integer value.
DateTime to Float	Converts date to number of days since 12/31/1899, time to a fraction of 24 hours.
Boolean to String	Converts any Boolean value to "*True*" or "*False*."

In other cases, conversions are not possible at all. In these cases, attempting a conversion also raises an exception.

You use a conversion function as you would use any method belonging to a component: append the function name to the end of the component name wherever it occurs in an assignment statement. Conversion always occurs before an actual assignment is made. For example, the following statement converts the value of *CustomersCustNo* to a string and assigns the string to the text of an edit control:

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

Conversely, the next statement assigns the text of an edit control to the *CustomersCustNo* field as an integer:

```
MyTableMyField.AsInteger := StrToInt(Edit1.Text);
```

An exception occurs if an unsupported conversion is performed at runtime.

Accessing field values with the default dataset method

See also

The preferred method for accessing a field's value is to use variants with the *FieldValues* property. For example, the following statement puts the value of an edit box into the *CustNo* field in the *Customers* table:

```
Customers.FieldValues['CustNo'] := Edit2.Text;
```

Accessing field values with a dataset's Fields property

See also

You can access the value of a field with the *Fields* property of the dataset component to which the field belongs. Accessing field values with a dataset's *Fields* property is useful when you need to iterate over a number of columns, or if your application works with tables that are not available to you at design time.

To use the *Fields* property you must know the order of and data types of fields in the dataset. You use an ordinal number to specify the field to access. The first field in a dataset is numbered 0. Field values must be converted as appropriate using the field component's conversion routine. For more information about field component conversion functions, see "Converting field values."

For example, the following statement assigns the current value of the seventh column (Country) in the *Customers* table to an edit control:

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

Conversely, you can assign a value to a field by setting the *Fields* property of the dataset to the desired field. For example:

```
begin
  Customers.Edit;
  Customers.Fields[6].AsString := Edit1.Text;
  Customers.Post;
end;
```

Accessing field values with a dataset's FieldByName method

See also

You can access the value of a field with a dataset's *FieldByName* method. This method is useful when you know the name of the field you want to access, but do not have access to the underlying table at design time.

To use *FieldByName*, you must know the dataset and name of the field you want to access. You pass the field's name as an argument to the method. To access or change the field's value, convert the result with the appropriate field component conversion function, such as *AsString* or *AsInteger*. For example, the following statement assigns the value of the *CustNo* field in the *Customers* dataset to an edit control:

```
Edit2.Text := Customers.FieldByName('CustNo').AsString;
```

Conversely, you can assign a value to a field:

```
begin
  Customers.Edit;
  Customers.FieldByName('CustNo').AsString := Edit2.Text;
  Customers.Post;
end;
```

Checking a field's current value

See also

If your application uses *TClientDataSet* or administers a dataset that is the source dataset for a *TProvider* component on an application server, and you encounter difficulties when updating records, you can use the *CurValue* property to examine the field value in the record causing problems. *CurValue* represents the current value of the field component including changes made by other users of the database.

Use *CurValue* to examine the value of a field when a problem occurs in posting a value to the database. If the current field value is causing a problem, such as a key violation, when posting the value to the database, an *OnReconcileError* occurs. In an *OnReconcileError* event handler, *NewValue* is the unposted value that caused the problem, *OldValue* is the value that was originally assigned to the field before any edits were made, and *CurValue* is the value that is currently assigned to the field. *CurValue* may differ from *OldValue* if another user changed the value of the field after *OldValue* was read.

Setting a default value for a field

See also

You can specify how a default value for a field should be calculated at runtime using the *DefaultExpression* property. *DefaultExpression* can be any valid SQL value expression that does not refer to field values. If the expression contains literals other than numeric values, they must appear in quotes. For example, a default value of noon for a time field would be

```
'12:00:00'
```

including the quotes around the literal value.

Working with constraints

See also

Field components can use SQL server constraints. In addition, your applications can create and use custom constraints that are local to your application. All constraints are rules or conditions that impose a limit on the scope or range of values that a field can store. The following sections describe working with constraints at the field component level.

The following topics are discussed in this section:

- Creating a custom constraint
- Using server constraints

Creating a custom constraint

See also

A custom constraint is not imported from the server like other constraints. It is a constraint that you declare, implement, and enforce in your local application. As such, custom constraints can be useful for offering a pre-validation enforcement of data entry, but a custom constraint cannot be applied against data received from or sent to a server application.

To create a custom constraint, set the *CustomConstraint* property to specify a constraint condition, and set *ConstraintErrorMessage* to the message to display when a user violates the constraint at runtime.

CustomConstraint is an SQL string that specifies any application-specific constraints imposed on the field's value. Set *CustomConstraint* to limit the values that the user can enter into a field. *CustomConstraint* can be any valid SQL search expression such as

```
x > 0 and x < 100
```

The name used to refer to the value of the field can be any string that is not a reserved SQL keyword, as long as it is used consistently throughout the constraint expression.

Custom constraints are imposed in addition to any constraints to the field's value that come from the server. To see the constraints imposed by the server, read the *ImportedConstraint* property.

Using server constraints

See also

Most production SQL databases use constraints to impose conditions on the possible values for a field. For example, a field may not permit NULL values, may require that its value be unique for that column, or that its values be greater than *0* and less than *150*. While you could replicate such conditions in your client applications, Delphi offers the *ImportedConstraint* property to propagate a server's constraints locally.

ImportedConstraint is a read-only property that specifies an SQL clause that limits field values in some manner. For example:

```
Value > 0 and Value < 100
```

Do not change the value of *ImportedConstraint*, except to edit nonstandard or server-specific SQL that has been imported as a comment because it cannot be interpreted by the database engine.

To add additional constraints on the field value, use the *CustomConstraint* property. Custom constraints are imposed in addition to the imported constraints. If the server constraints change, the value of *ImportedConstraint* also changed but constraints introduced in the *CustomConstraint* property persist.

Removing constraints from the *ImportedConstraint* property will not change the validity of field values that violate those constraints. Removing constraints results in the constraints being checked by the server instead of locally. When constraints are checked locally, the error message supplied as the *ConstraintErrorMessage* property is displayed when violations are found, instead of displaying an error message from the server.

Working with object fields

See also

Object field (*TObjectField*) descendants support the field types ADT (Abstract Data Type), Array, DataSet, and Reference. All of these field types either contain or reference child fields or other data sets.

ADT fields and reference fields map to fields that contain child fields. An ADT field contains child fields, which themselves can be any scalar or object type. An array field contains an array of child fields, all of the same type.

Dataset and reference fields map to fields that access other data sets. A dataset field provides access to a nested data set and a reference field stores a pointer (reference) to another persistent object (ADT).

Component name	Purpose
TADTField	Represents an ADT (Abstract Data Type) field.
TArrayField	Represents an array field.
TDataSetField	Represents a field that contains a nested data set reference.
TReferenceField	Represents a REF field, a pointer to an ADT.

When you add fields with the Fields editor to a dataset that contains object fields, persistent object fields of the correct type are automatically created for you. Adding persistent object fields to a dataset automatically sets the dataset's *ObjectView* property to *True*, which instructs the fields to be stored hierarchically rather than flattened out.

The following properties are common to all object field descendants and provide the functionality to handle child fields and datasets.

Property	Purpose
Fields	Contains the child fields belonging to the object field.
ObjectType	Classification of the object field.
FieldCount	Number of child fields belonging to the object field.
FieldValues	Provides access to the values of the child fields of the object field.

The following topics are discussed in this section:

- Working with ADT fields
- Working with array fields
- Working with dataset fields
- Working with reference fields

Displaying ADT and array fields

Both ADT and array fields contain child fields that can be displayed through data-aware controls. Data-Aware controls such as *TDBEdit* and *TDBGrid* automatically display ADT and array field types.

Data-aware controls with a *DataField* property automatically displays any ADT and array fields and their child fields in the drop-down list. When an ADT or array field is bound to a data-aware control, the child fields appear in an uneditable comma delimited string in the control. A child field is bound to the control as a normal data field.

A *TDBGrid* control displays ADT and array field data differently, depending on the value of the dataset's *ObjectView* property. When *ObjectView* is False, each child field appears in a single column. When *ObjectView* is True, an ADT or array field can be expanded and collapsed by clicking on the arrow in the title bar of the column. When the field is expanded, each child field appears in its own column and title bar, all below the title bar of the ADT or array itself. When the ADT or array is collapsed, only one column appears with an uneditable comma delimited string containing the child fields.

Working with ADT fields

ADTs are user-defined types created on the server, and are similar to structures. An ADT can contain most scalar field types, array fields, reference fields, and nested ADTs.

Accessing ADT field values

There are a variety of ways to access the data in ADT field types. Creating and using persistent fields is strongly recommended. The following examples assign a child field value to an edit box called *CityEdit*, and uses the following ADT structure,

```
Address
  Street
  City
  State
  Zip
```

and the following persistent fields created for the *Customer* table component,

```
CustomerAddress: TADTField;
CustomerAddrStreet: TStringField;
CustomerAddrCity: TStringField;
CustomerAddrState: TStringField;
CustomerAddrZip: TStringField;
```

This line of code uses a persistent field and demonstrates the recommended method of accessing data in ADT fields.

```
CityEdit.Text := CustomerAddrCity.AsString;
```

The following code examples require that the dataset's *ObjectView* property be set to *True* in order to compile. They don't require persistent fields.

This example uses a fully qualified name with the *FieldByName* method on the dataset.

```
CityEdit.Text := Customer.FieldByName('Address.City').AsString;
```

You can access the value of a child field with the TADTField's *FieldValues* property. *FieldValues* accepts and returns a *Variant*, so it can handle and convert fields of any type. The index parameter takes an integer value which specifies the offset of the field. It is also the default property on *TObjectField*, and can therefore be omitted. For example,

```
CityEdit.Text := TADTField(Customer.FieldByName('Address'))[1];
```

which is the same as,

```
CityEdit.Text := TADTField(Customer.FieldByName('Address')).FieldValues[1];
```

This code uses the *Fields* property of the TADTField component.

```
CityEdit.Text := TADTField(Customer.FieldByName('Address')).Fields[1].AsString;
```

This code uses the *Fields* property of the *TADTField* component with *FieldByName* of both the dataset and the *TFields* object.

```
CityEdit.Text :=
TADTField(Customer.FieldByName('Address')).Fields.FieldByName('City').AsString;
```

As you can see from this last example, accessing the field's data through persistent fields is much simpler. These additional access methods are primarily useful when the structure of the database table is not fixed or known at design time.

ADT field values can also be accessed with a dataset's FieldValues property:

```
Customer.Edit;
Customer['Address.City'] := CityEdit.Text;
Customer.Post;
```

The next statement reads a string value from the *City* child field of the ADT field Address into an edit box:

```
CityEdit.Text := Customer['Address.City'];
```

Note: The dataset's *ObjectView* property can be either *True* or *False* for these lines of code to compile.

Working with Array fields

Array fields consist of a set of fields of the same type. The field types can be scalar (e.g. float, string), or non-scalar (an ADT), but an array field of arrays is not permitted. The *SparseArrays* property of *TDataSet* determines whether a unique *TField* object is created for each element of the array field.

Accessing array field values

There are a variety of ways to access the data in array field types. The following example populates a list box with all of the non-null array elements.

```
var
  OrderDates: TArrayField;
  I: Integer;
begin
  for I := 0 to OrderDates.Size - 1 do
  begin
    if OrderDates.Fields[I].IsNull then Break;
    OrderDateListBox.Items.Add(OrderDates[I]);
  end;
end;
```

The following examples assign a child field value to an edit box called *TelEdit*, and uses the array *TelNos_Array*, which is a six element array of strings. The following persistent fields created for the *Customer* table component are used by the following examples:

```
CustomerTelNos_Array: TArrayField;
CustomerTelNos_Array0: TStringField;
CustomerTelNos_Array1: TStringField;
CustomerTelNos_Array2: TStringField;
CustomerTelNos_Array3: TStringField;
CustomerTelNos_Array4: TStringField;
CustomerTelNos_Array5: TStringField;
```

This line of code uses a persistent field to assign an array element value to an edit box.

```
TelEdit.Text := CustomerTelNos_Array0.AsString;
```

The following code examples require that the dataset's *ObjectView* property be set to *True* in order to compile. They don't require persistent fields.

You can access the value of a child field with the dataset's *FieldValues* property. FieldValues accepts and returns a *Variant*, so it can handle and convert fields of any type. For example,

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array'))[1];
```

which is the same as,

```
TelEdit.Text := TArrayField(Customer.FieldByName('TelNos_Array')).FieldValues[1];
```

This next code example uses the *Fields* property of the *TArrayField* component.

```
TelEdit.Text :=
TArrayField(Customer.FieldByName('TelNos_Array')).Fields[1].AsString;
```

Working with DataSet fields

See also

Dataset fields provide access to data stored in a nested dataset. The NestedDataSet property references the nested dataset. The data in the nested dataset is then accessed through the field objects of the nested dataset.

Displaying dataset fields

TDBGrid controls enable the display of data stored in data set fields. In a *TDBGrid* control, a dataset field is indicated in each cell of a dataset column with a "(DataSet)", and at runtime an ellipsis button also exists to the right. Clicking on the ellipsis brings up a new form with a grid displaying the dataset associated with the current record's dataset field. This form can also be brought up programmatically with the DB grid's *ShowPopupEditor* method. For example, if the seventh column in the grid represents a dataset field, the following code will display the dataset associated with that field for the current record.

```
DBGrid1.ShowPopupEditor(DBGrid1.Columns[7]);
```

Accessing data in a nested dataset

A dataset field is not normally bound directly to a data aware control. Rather, since a nested data set is just that, a data set, the means to get at its data is via a *TDataSet* descendant. This particular *TDataSet* descendant is *TNestedTable*, and it provides the specific functionality needed to access data stored in nested datasets. Once a *TDataSetField* is associated with a dataset field, persistent fields can be created for the fields of the nested dataset.

To access the data in a dataset field you first create a persistent *TDataSetField* object by invoking the table's Fields editor, and then link to this field using the *DataSetField* property on a *TNestedTable* or *TClientDataSet* object. If the nested dataset field for the current record is assigned, the nested dataset will contain records with the nested data; otherwise, the nested dataset will be empty.

Before inserting records into a nested dataset, you should be sure to post the corresponding record in the master table, if it has just been inserted. If the inserted record is not posted, it will be automatically posted before the nested dataset posts.

Working with Reference fields

See also

Reference fields store a pointer or reference to another ADT object. This ADT object is a single record of another object table. Reference fields always refer to a single record in a dataset (object table). The data in the referenced object is actually returned in a nested dataset, but can also be accessed via the *Fields* property on the *TReferenceField*.

Displaying reference fields

In a *TDBGrid* control a reference field is designated in each cell of the dataset column, with (Reference) and, at runtime, an ellipsis button to the right. At runtime, clicking on the ellipsis brings up a new form with a grid displaying the object associated with the current record's reference field.

This form can also be brought up programmatically with the DB grid's *ShowPopupEditor* method. For example, if the seventh column in the grid represents a reference field, the following code will display the object associated with that field for the current record.

```
DBGrid1.ShowPopupEditor(DBGrid1.Columns[7]);
```

Accessing data in a reference field

To access the data in a reference field you first create a persistent *TDataSetField*, and then link to this field using the *DataSetField* property on a *TNestedTable* or *TClientDataSet*. If the reference is assigned, the reference will contain a single record with the referenced data. If the reference is null, the reference will be empty.

The following examples are equivalent and assign data from the reference field *CustomerRefCity* to an edit box called *CityEdit*:

```
CityEdit.Text := CustomerRefCity.Fields[1].AsString;
CityEdit.Text := CustomerRefCity.NestedDataSet.Fields[1].AsString;
```

When data in a reference field is edited, it is actually the referenced data that is modified.

To assign a reference field, you need to first use a SELECT statement to select the reference from the table, and then assign. For example:

```
var
  AddressQuery: TQuery;
  CustomerAddressRef: TReferenceField;
begin
  AddressQuery.SQL.Text := 'SELECT REF(A) FROM AddressTable A WHERE A.City = ''San
Francisco''';
  AddressQuery.Open;
  CustomerAddressRef.Assign(AddressQuery.Fields[0]);
end;
```

Working with tables

Topic groups

This section describes how to use the *TTable* dataset component in your database applications. A table component encapsulates the full structure of and data in an underlying database table. A table component inherits many of its fundamental properties and methods from *TDataSet*, *TBDEDataSet*, and *TDBDataSet*. Therefore, you should be familiar with the general discussion of datasets in "Understanding datasets," and the BDE-specific discussion of datasets in "Using BDE-enabled datasets," before reading about the unique properties and methods of table components discussed here.

The following topics are discussed in this section:

- Using table components
- Setting up a table component
- Controlling read/write access to a table
- Searching for records
- Sorting records
- Working with a subset of data
- Deleting all records in a table
- Deleting a table
- Renaming a table
- Creating a table
- Importing data from another table
- Synchronizing tables linked to the same database table
- Creating master/detail forms

Using table components

Topic groups See also

A table component gives you access to every row and column in an underlying database table, whether it is from Paradox, dBASE, Access, FoxPro, an ODBC-compliant database, or an SQL database on a remote server, such as InterBase, Sybase, or SQL Server.

You can view and edit data in every column and row of a table. You can work with a range of rows in a table, and you can filter records to retrieve a subset of all records in a table based on filter criteria you specify. You can search for records, copy, rename, or delete entire tables, and create master/detail relationships between tables.

Note: A table component always references a single database table. If you need to access multiple tables with a single component, or if you are only interested in a subset of rows and columns in one or more tables, you should use a query component instead of a table component. For more information about query components, see "Working with queries."

Setting up a table component

Topic groups See also

The following steps are general instructions for setting up a table component at design time. There may be additional steps you need to tailor a table's properties to the requirements of your application.

- To create a table component,

1. Place a table component from the Data Access page of the Component palette in a data module or on a form, and set its *Name* property to a unique value appropriate to your application.
2. Set the *DatabaseName* of the component to the name of the database to access.
3. Set the *TableName* property to the name of the table in the database. You can select tables from the drop-down list if the *DatabaseName* property is already specified.
4. Place a data source component in the data module or on the form, and set its *DataSet* property to the name of the table component. The data source component is used to pass a result set from the table to data-aware components for display.

- To access the data encapsulated by a table component,

1. Place a data source component from the Data Access page of the Component palette in the data module or form, and set its *DataSet* property to the name of the table component.
2. Place a data-aware control, such as *TDBGrid*, on a form, and set the control's *DataSource* property to the name of the data source component placed in the previous step.
3. Set the *Active* property of the table component to *True*.

The following topics are discussed in this section:

- Specifying a database location
- Specifying a table name
- Specifying the table type for local tables
- Opening and closing a table

Specifying a database location

Topic groups See also

The *DatabaseName* property specifies where the table component looks for a database table. For Paradox and dBASE, *DatabaseName* can be a Borland Database Engine (BDE) alias, or an explicit directory path. For SQL tables, *DatabaseName* must be a BDE alias.

The advantage of using BDE aliases in all cases is that you can change the data source for an entire application by simply changing the alias definition in the SQL Explorer. To change the alias definition using SQL explorer, right click the SQL explorer and select Rename. This displays the BDE Administration Tool. For more information about setting and using BDE aliases, see the online help for the SQL Explorer.

To set the *DatabaseName* property,

1. Set the table's *Active* property to *False* if necessary.
2. Specify the BDE alias or directory path in the *DatabaseName* property.

Tip: If your application uses database components to control database transactions, *DatabaseName* can be set to a local alias defined for the database component instead. For more information about database components, see "Connecting to databases."

Specifying a table name

Topic groups See also

The *TableName* property specifies the table in a database to access with the table component. To specify a table, follow these steps:

1. Set the table's *Active* property to *False*, if necessary.
2. Set the *DatabaseName* property to a BDE alias or directory path. For more information about setting *DatabaseName*, see "Specifying a database location."
3. Set the *TableName* property to the table to access. At design time you can choose from valid table names in the drop-down list for the *TableName* property in the Object Inspector. At runtime, you must specify a valid name in code.

Once you specify a valid table name, you can set the table component's *Active* property to *True* to connect to the database, open the table, and display and edit data.

At runtime, you can set or change the table associated with a table component by:

- Setting *Active* to *False*.
- Assigning a valid table name to the *TableName* property.

For example, the following code changes the table name for the *OrderOrCustTable* table component based on its current table name:

```
with OrderOrCustTable do
begin
  Active := False; {Close the table}
  if TableName = 'CUSTOMER.DB' then
    TableName := 'ORDERS.DB'
  else
    TableName := 'CUSTOMER.DB';
  Active := True; {Reopen with a new table}
end;
```

Specifying the table type for local tables

Topic groups See also

If an application accesses Paradox, dBASE, FoxPro, or comma-delimited ASCII text tables, then the BDE uses the *TableType* property to determine the table's type (its expected structure). *TableType* is not used when an application accesses SQL-based tables on database servers.

By default *TableType* is set to *ttDefault*. When *TableType* is *ttDefault*, the BDE determines a table's type from its file-name extension. The following table summarizes the file name extensions recognized by the BDE and the assumptions it makes about a table's type:

Extension	Table type
No file extension	Paradox
.DB	Paradox
.DBF	dBASE
.TXT	ASCII text

If your local Paradox, dBASE, and ASCII text tables use the file extensions as described in the previous table, then you can leave *TableType* set to *ttDefault*. Otherwise, your application must set *TableType* to indicate the correct table type. The following table indicates the values you can assign to *TableType*:

Value	Table type
ttDefault	Table type determined automatically by the BDE
ttParadox	Paradox
ttDBase	dBASE
ttFoxPro	FoxPro
ttASCII	Comma-delimited ASCII text

Opening and closing a table

Topic groups See also

To view and edit a table's data in a data-aware control such as *TDBGrid*, open the table. There are two ways to open a table. You can set its *Active* property to *True*, or you can call its *Open* method. Opening a table puts it into *dsBrowse* state and displays data in any active controls associated with the table's data source.

To end display and editing of data, or to change the values for a table component's fundamental properties (e.g., *DatabaseName*, *TableName*, and *TableType*), first post or discard any pending changes. If cached updates are enabled, call the *ApplyUpdates* method to write the posted changes to the database. Finally, close the table.

There are two ways to close a table. You can set its *Active* property to *False*, or you can call its *Close* method. Closing a table puts the table into *dsInactive* state. Active controls associated with the table's data source are cleared.

Controlling read/write access to a table

Topic groups See also

By default when a table is opened, it requests read and write access for the underlying database table. Depending on the characteristics of the underlying database table, the requested write privilege may not be granted (for example, when you request write access to an SQL table on a remote server and the server restricts the table's access to read only).

There are three properties for table components that can affect an application's read and write access to a table: *CanModify*, *ReadOnly*, and *Exclusive*.

CanModify is a read-only property that specifies whether or not a table component is permitted read/write access to the underlying database table. After you open a table at runtime, your application can examine *CanModify* to test whether or not the table has write access. If *CanModify* is *False*, the application cannot write to the database. If *CanModify* is *True*, your application can write to the database provided that the table's *ReadOnly* property is *False*.

ReadOnly determines whether or not a user can both view and edit data. When *ReadOnly* is *False* (the default), a user can both view and edit data. To restrict a user to viewing data, set *ReadOnly* to *True* before opening a table.

Exclusive controls whether or not an application gains sole read/write access to a Paradox, dBASE, or FoxPro table. To gain sole read/write access for these table types, set the table component's *Exclusive* property to *True* before opening the table. If you succeed in opening a table for exclusive access, other applications cannot read data from or write data to the table. Your request for exclusive access is not honored if the table is already in use when you attempt to open it.

The following statements open a table for exclusive access:

```
CustomersTable.Exclusive := True; {Set request for exclusive lock}
CustomersTable.Active := True; {Now open the table}
```

Note: You can attempt to set *Exclusive* on SQL tables, but some servers may not support exclusive table-level locking. Others may grant an exclusive lock, but permit other applications to read data from the table. For more information about exclusive locking of database tables on your server, see your server documentation.

Searching for records

Topic groups See also

You can search for specific records in a table in various ways. The most flexible and preferred way to search for a record is to use the generic search methods *Locate* and *Lookup*. These methods enable you to search on any type of fields in any table, whether or not they are indexed or keyed.

- *Locate* finds the first row matching a specified set of criteria and moves the cursor to that row.
- *Lookup* returns values from the first row that matches a specified set of criteria, but does not move the cursor to that row.

You can use *Locate* and *Lookup* with any kind of dataset, not just *TTable*. For a complete discussion of *Locate* and *Lookup*, see "Understanding datasets."

Table components also support the *Goto* and *Find* methods. While these methods are documented here to allow you to work with legacy applications, you should always use *Lookup* and *Locate* in your new applications. You may see performance gains in existing applications if you convert them to use the new methods.

The following topics are discussed in this section:

- Searching for records based on indexed fields
- Specifying the current record after a successful search
- Searching on partial keys
- Searching on alternate indexes

Searching for records based on indexed fields

Topic groups See also

Table components support a set of *Goto* search methods for backward compatibility. *Goto* methods enable you to search for a record based on indexed fields, referred to as a *key*, and make the first record found the new current record.

For Paradox and dBASE tables, the key must always be an index, which you can specify in a table component's *IndexName* property. For SQL tables, the key can also be a list of fields you specify in the *IndexFieldNames* property. You can also specify a field list for Paradox or dBASE tables, but the fields must have indexes defined on them. For more information about *IndexName* and *IndexFieldNames*, see "Searching on alternate indexes."

Tip: To search on nonindexed fields in a Paradox or dBASE table, use *Locate*. Alternatively, you can use a *TQuery* component and a SELECT statement to search on nonindexed fields in Paradox and dBASE fields. For more information about *TQuery*, see "Working with queries."

The following table summarizes the six related *Goto* and *Find* methods an application can use to search for a record:

Method	Purpose
EditKey	Preserves the current contents of the search key buffer and puts the table into *dsSetKey* state so your application can modify existing search criteria prior to executing a search.
FindKey	Combines the *SetKey* and *GotoKey* methods in a single method.
FindNearest	Combines the *SetKey* and *GotoNearest* methods in a single method.
GotoKey	Searches for the first record in a dataset that exactly matches the search criteria, and moves the cursor to that record if one is found.
GotoNearest	Searches on string-based fields for the closest match to a record based on partial key values, and moves the cursor to that record.
SetKey	Clears the search key buffer and puts the table into *dsSetKey* state so your application can specify new search criteria prior to executing a search.

GotoKey and *FindKey* are Boolean functions that, if successful, move the cursor to a matching record and return *True*. If the search is unsuccessful, the cursor is not moved, and these functions return *False*.

GotoNearest and *FindNearest* always reposition the cursor either on the first exact match found or, if no match is found, on the first record that is greater than the specified search criteria.

The following topics are discussed in this section:

- Executing a search with Goto methods
- Executing a search with Find methods

Executing a search with Goto methods

Topic groups See also

To execute a search using *Goto* methods, follow these general steps:

1. Specify the index to use for the search in the *IndexName* property, if necessary. (For SQL tables, list the fields to use as a key in *IndexFieldNames* instead.) If you use a table's primary index, you do not need to set these properties.
2. Open the table.
3. Put the table in *dsSetKey* state with *SetKey*.
4. Specify the value(s) to search on in the *Fields* property. Fields is a string list that you index into with ordinal numbers corresponding to columns. The first column number in a table is 0.
5. Search for and move to the first matching record found with *GotoKey* or *GotoNearest*.

For example, the following code, attached to a button's *OnClick* event, moves to the first record containing a field value that exactly matches the text in an edit box on a form:

```
procedure TSearchDemo.SearchExactClick(Sender: TObject);
begin
  Table1.SetKey;
  Table1.Fields[0].AsString := Edit1.Text;
  if not Table1.GotoKey then
    ShowMessage('Record not found');
end;
```

GotoNearest is similar. It searches for the nearest match to a partial field value. It can be used only for string fields. For example,

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

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

Executing a search with Find methods

Topic groups See also

To execute a search using *Find* methods, follow these general steps:

1. Specify the index to use for the search in the *IndexName* property, if necessary. (For SQL tables, list the fields to use as a key in *IndexFieldNames* instead.) If you use a table's primary index, you do not need to set these properties.
2. Open the table.
3. Search for and move to the first or nearest record with *FindKey* or *FindNearest*. Both methods take a single parameter, a comma-delimited list of field values, where each value corresponds to an indexed column in the underlying table.

Note: *FindNearest* can only be used for string fields.

Specifying the current record after a successful search

Topic groups See also

By default, a successful search positions the cursor on the first record that matches the search criteria. If you prefer, you can set the *KeyExclusive* property for a table component to *True* to position the cursor on the next record after the first matching record.

By default, *KeyExclusive* is *False*, meaning that successful searches position the cursor on the first matching record.

Searching on partial keys

Topic groups See also

If a table has more than one key column, and you want to search for values in a sub-set of that key, set *KeyFieldCount* to the number of columns on which you are searching. For example, if a table has a three column primary key, and you want to search for values in just the first column, set *KeyFieldCount* to 1.

For tables with multiple-column keys, you can search only for values in contiguous columns, beginning with the first. For example, for a three-column key you can search for values in the first column, the first and second, or the first, second, and third, but not just the first and third.

Searching on alternate indexes

Topic groups See also

If you want to search on an index other than the primary index for a table, then you must specify the name of the index to use in the *IndexName* property for the table. For example, if the CUSTOMER table had a secondary index named “CityIndex” on which you wanted to search for a value, you need to set the value of the table’s *IndexName* property to “CityIndex”:

```
Table1.Close;
Table1.IndexName := 'CityIndex';
Table1.Open;
Table1.SetKey;
Table1['City'] := Edit1.Text;
Table1.GotoNearest;
```

Instead of specifying an index name, you can list fields to use as a key in the *IndexFieldNames* property. For Paradox and dBASE tables, the fields you list must be indexed, or an exception is raised when you execute the search. For SQL tables, the fields you list need not be indexed.

Repeating or extending a search

Each time you call *SetKey* or *FindKey* it clears any previous values in the *Fields* property. If you want to repeat a search using previously set fields, or you want to add to the fields used in a search, call *EditKey* in place of *SetKey* and *FindKey*. For example, if the “CityIndex” index includes both the *City* and *Country* fields, to find a record with a specified company name in a specified city, use the following code:

```
Table1.EditKey;
Table1['Country'] := Edit2.Text;
Table1.GotoNearest;
```

Sorting records

Topic groups See also

An index determines the display order of records in a table. In general, records appear in ascending order based on a primary index (for dBASE tables without a primary index, sort order is based on physical record order). This default behavior does not require application intervention. If you want a different sort order, however, you must specify either

- An alternate index.
- A list of columns on which to sort (SQL only).

Specifying a different sort order requires the following steps:

1. Determining available indexes.
2. Specifying the alternate index or column list to use.

The following topics are discussed in this section:

- Retrieving a list of available indexes with GetIndexNames
- Specifying an alternative index with IndexName
- Specifying sort order for SQL tables
- Examining the field list for an index

Retrieving a list of available indexes with GetIndexNames

Topic groups See also

At runtime, your application can call the *GetIndexNames* method to retrieve a list of available indexes for a table. *GetIndexNames* returns a string list containing valid index names. For example, the following code determines the list of indexes available for the *CustomersTable* dataset:

```
var
  IndexList: TList;
...
CustomersTable.GetIndexNames(IndexList);
```

Note: For Paradox tables, the primary index is unnamed, and is therefore not returned by *GetIndexNames*. If you need to return to using a primary index on a Paradox table after using an alternative index, set the table's *IndexName* property to a null string.

Specifying an alternative index with IndexName

Topic groups See also

To specify that a table should be sorted using an alternative index, specify the index name in the table component's *IndexName*property. At design time, you can specify this name in the Object Inspector, and at runtime you can access the property in your code. For example, the following code sets the index for *CustomersTable* to *CustDescending*:

```
CustomersTable.IndexName := 'CustDescending';
```

For information on specifying dBASE index files, see "Specifying a dBASE index file."

Specifying a dBASE index file

Topic groups

For dBASE tables that use non-production indexes, set the *IndexFiles*property to the name of the index file(s) to use before you set *IndexName*. At design time, you can click the ellipsis button in the *IndexFiles* property value in the Object Inspector to invoke the Index Files editor.

To see a list of available index files, choose Add, and select one or more index files from the list. A dBASE index file can contain multiple indexes. To select an index from the index file, select the index name from the *IndexName* drop-down list in the Object Inspector. You can also specify multiple indexes in the file by entering desired index names, separated by semicolons.

You can also set *IndexFiles* and *IndexName* at runtime. For example, the following code sets the *IndexFiles* for the *AnimalsTable* table component to ANIMALS.MDX, and then sets *IndexName* to NAME:

```
AnimalsTable.IndexFiles := 'ANIMALS.MDX';
AnimalsTable.IndexName := 'NAME';
```

Specifying sort order for SQL tables

Topic groups See also

In SQL, sort order of rows is determined by the ORDER BY clause. You can specify the index used by this clause either with the

- *IndexName*property, to specify an existing index, or
- *IndexFieldNames* property, to create a pseudo-index based on a subset of columns in the table.

IndexName and *IndexFieldNames* are mutually exclusive. Setting one property clears values set for the other. To use *IndexName*, see "Searching on alternate indexes."

For information on specifying fields using the *IndexFieldNames* property, see "Specifying fields with IndexFieldNames."

Specifying fields with IndexFieldNames

Topic groups

IndexFieldNames is a string list property. To specify a sort order, list each column name to use in the order it should be used, and delimit the names with semicolons. Sorting is by ascending order only. Case-sensitivity of the sort depends on the capabilities of your server. See your server documentation for more information.

The following code sets the sort order for *PhoneTable* based on *LastName*, then *FirstName*:

```
PhoneTable.IndexFieldNames := 'LastName;FirstName';
```

Note: If you use *IndexFieldNames* on Paradox and dBASE tables, Delphi attempts to find an index that uses the columns you specify. If it cannot find such an index, it raises an exception.

Examining the field list for an index

Topic groups See also

When your application uses an index at runtime, it can examine the

- *IndexFieldCount* property, to determine the number of columns in the index.
- *IndexFields* property, to examine a list of column names that comprise the index.

IndexFields is a string list containing the column names for the index. The following code fragment illustrates how you might use *IndexFieldCount* and *IndexFields* to iterate through a list of column names in an application:

```
var
  I: Integer;
  ListOfIndexFields: array[0 to 20} of string;
begin
with CustomersTable do
  begin
  for I := 0 to IndexFieldCount - 1 do
     ListOfIndexFields[I] := IndexFields[I];
  end;
end;
```

Note: *IndexFieldCount* is not valid for a base table opened on an expression index.

Working with a subset of data

Topic groups See also

Production tables can be huge, so applications often need to limit the number of rows with which they work. For table components, there are two ways to limit records used by an application: ranges and filters. Filters can be used with any kind of dataset, including *TTable*, *TQuery*, and *TStoredProc* components. Because they apply to all datasets, you can find a full discussion of using filters in "Understanding datasets."

Ranges only apply to *TTable* components. Despite their similarities, ranges and filters have different uses. The following section discusses the differences between ranges and filters and then discusses how to use ranges.

The following topics are discussed in this section:

- Understanding the differences between ranges and filters
- Creating and applying a new range
- Modifying a range

Understanding the differences between ranges and filters

Topic groups See also

Both ranges and filters restrict visible records in a table to a subset of all available records, but the way they do so differs. A range is a set of contiguously indexed records that fall between specified boundary values. For example, in an employee database indexed on last name, you might apply a range to display all employees whose last names are greater than “Jones” and less than “Smith”. Because ranges depend on indexes, ranges can only be applied to indexed Paradox and dBASE tables (for SQL tables, ranges can be applied to any fields you list in the *IndexFieldNames* property). Ranges can only be ordered based on existing indexes.

A filter, on the other hand, is any set of contiguous and noncontiguous records that share specified data points, regardless of indexing. For example, you might filter an employee database to display all employees who live in California and who have worked for the company for five or more years. While filters make use of indexes if they apply, filters are not dependent on them. Filters are applied record-by-record as an application scrolls through a dataset.

In general, filters are more flexible than ranges. Ranges, however, can be more efficient when datasets are very large and the records of interest to an application are already blocked in contiguously indexed groups. For very large datasets, it may be still more efficient to use a query component to select data for viewing and editing. For more information about using filters, see "Understanding datasets.” For more information about using queries, see "Working with queries.”

Creating and applying a new range

Topic groups See also

The process for creating and applying a range involves these general steps:

1. Putting the dataset into *dsSetKey* state and setting the starting index value for the range.
2. Setting the ending index value for the range.
3. Applying the range to the dataset.

The following topics are discussed in this section:

- Setting the beginning of a range
- Setting the end of a range
- Setting start- and end-range values
- Specifying a range based on partial keys
- Including or excluding records that match boundary values
- Applying a range
- Canceling a range

Setting the beginning of a range

Topic groups See also

Call the *SetRangeStart* procedure to put the dataset into *dsSetKey* state and begin creating a list of starting values for the range. Once you call *SetRangeStart*, subsequent assignments to the *Fields* property are treated as starting index values to use when applying the range. Fields specified must be indexed fields when using Paradox and dBASE.

For example, suppose your application uses a table component named *Customers*, linked to the CUSTOMER table, and that you have created persistent field components for each field in the *Customers* dataset. CUSTOMER is indexed on its first column (*CustNo*). A form in the application has two edit components named *StartVal* and *EndVal*, used to specify start and ending values for a range. If so, the following code could be used to create and apply a range:

```
with Customers do
begin
  SetRangeStart;
  FieldByName('CustNo') := StartVal.Text;
  SetRangeEnd;
  if EndVal.Text <> '' then
    FieldByName('CustNo') := EndVal.Text;
  ApplyRange;
end
```

This code checks that the text entered in *EndVal* is not null before assigning any values to *Fields*. If the text entered for *StartVal* is null, then all records from the beginning of the 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.

For a multi-column index, you can specify a starting value for all or some fields in the index. If you do not supply a value for a field used in the index, a null value is assumed when you apply the range. If you set more values than there are fields in the index, the extra fields are ignored when computing the range.

To finish specifying the start of a range, call *SetRangeEnd* or *ApplyRange*. These methods are discussed in the following sections.

Tip: To start at the beginning of the dataset, omit the call to *SetRangeStart*.

You can also set the starting (and ending) values for a range with a call to the *SetRange* procedure. For more information about *SetRange*, see "Setting start- and end-range values."

Setting the end of a range

Topic groups See also

Call the *SetRangeEnd* procedure to put the dataset into *dsSetKey* state and start creating a list of ending values for the range. Once you call *SetRangeEnd*, subsequent assignments to the *Fields* property are treated as ending index values to use when applying the range. Fields specified must be indexed fields for Paradox and dBASE tables.

Note: Always specify the ending values for a range, even if you want a range to end on the last record in the dataset. If you do not provide ending values, Delphi assumes the ending value of the range is a null value. A range with null ending values is always empty.

The easiest way to assign ending values is to call the *FieldByName* method. For example,

```
with Table1 do
begin
  SetRangeStart;
  FieldByName('LastName') := Edit1.Text;
  SetRangeEnd;
  FieldByName('LastName') := Edit2.Text;
  ApplyRange;
end;
```

For a multi-column index, you can specify a starting value for all or some fields in the index. If you do not supply a value for a field used in the index, a null value is assumed when you apply the range. If you set more values than there are fields in the index, an exception is raised.

To finish specifying the end of a range, call *ApplyRange*. For more information about applying a range, see "Applying a range."

Note: You can also set the ending (and starting) values for a range with a call to the *SetRange* procedure. *SetRange* is described in the next section.

Setting start- and end-range values

Topic groups See also

Instead of using separate calls to *SetRangeStart* and *SetRangeEnd* to specify range boundaries, you can call the *SetRange* procedure to put the dataset into *dsSetKey* state and set the starting and ending values for a range with a single call.

SetRange takes two constant array parameters: a set of starting values, and a set of ending values. For example, the following statements establish a range based on a two-column index:

```
SetRange([Edit1.Text, Edit2.Text], [Edit3.Text, Edit4.Text]);
```

For a multi-column index, you can specify starting and ending values for all or some fields in the index. If you do not supply a value for a field used in the index, a null value is assumed when you apply the range. To omit a value for the first field in an index, and specify values for successive fields, pass a null value for the omitted field. If you set more values than there are fields in the index, the extra fields are ignored when computing the range.

Note: Always specify the ending values for a range, even if you want a range to end on the last record in the dataset. If you do not provide ending values, Delphi assumes the ending value of the range is a null value. A range with null ending values is always empty because the starting range is greater than or equal to the ending range.

Specifying a range based on partial keys

Topic groups See also

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

```
Table1.SetRangeStart;
Table1['LastName'] := 'Smith';
Table1.SetRangeEnd;
Table1['LastName'] := 'Zzzzzz';
Table1.ApplyRange;
```

This code includes all records in a range where *LastName* is greater than or equal to “Smith.” The value specification could also be:

```
Table1['LastName'] := 'Sm';
```

This statement includes records that have *LastName* greater than or equal to “Sm.” The following statement includes records with a *LastName* greater than or equal to “Smith” and a *FirstName* greater than or equal to “J”:

```
Table1['LastName'] := 'Smith';
Table1['FirstName'] := 'J';
```

Including or excluding records that match boundary values

Topic groups See also

By default, a range includes all records that are greater than or equal to the specified starting range, and less than or equal to the specified ending range. This behavior is controlled by the *KeyExclusive* property. *KeyExclusive* is *False* by default.

If you prefer, you can set the *KeyExclusive* property for a table component to *True* to exclude records equal to ending range. For example,

```
KeyExclusive := True;
Table1.SetRangeStart;
Table1['LastName'] := 'Smith';
Table1.SetRangeEnd;
Table1['LastName'] := 'Tyler';
Table1.ApplyRange;
```

This code includes all records in a range where *LastName* is greater than or equal to “Smith” and less than “Tyler”.

Applying a range

Topic groups See also

The *SetRange* methods described in the preceding sections establish the boundary conditions for a range, but do not put it into effect. To make a range take effect, call the *ApplyRange* procedure. *ApplyRange* immediately restricts a user's view of and access to data in the specified subset of the dataset.

Canceling a range

Topic groups See also

The *CancelRange* method ends application of a range and restores access to the full dataset. Even though canceling a range restores access to all records in the dataset, the boundary conditions for that range are still available so that you can reapply the range at a later time. Range boundaries are preserved until you provide new range boundaries or modify the existing boundaries. For example, the following code is valid:

```
...
Table1.CancelRange;
...
{later on, use the same range again. No need to call SetRangeStart, etc.}
Table1.ApplyRange;
...
```

Modifying a range

Topic groups See also

Two functions enable you to modify the existing boundary conditions for a range: *EditRangeStart*, for changing the starting values for a range; and *EditRangeEnd*, for changing the ending values for the range.

The process for editing and applying a range involves these general steps:

1. Putting the dataset into *dsSetKey* state and modifying the starting index value for the range.
2. Modifying the ending index value for the range.
3. Applying the range to the dataset.

You can modify either the starting or ending values of the range, or you can modify both boundary conditions. If you modify the boundary conditions for a range that is currently applied to the dataset, the changes you make are not applied until you call *ApplyRange* again.

The following topics are discussed in this section:

- Editing the start of a range
- Editing the end of a range

Editing the start of a range

Topic groups See also

Call the *EditRangeStart* procedure to put the dataset into *dsSetKey* state and begin modifying the current list of starting values for the range. Once you call *EditRangeStart*, subsequent assignments to the *Fields* property overwrite the current index values to use when applying the range. Fields specified must be indexed fields when using Paradox and dBASE.

Tip: If you initially created a start range based on a partial key, you can use *EditRangeStart* to extend the starting value for a range. For more information about ranges based on partial keys, see "Specifying a range based on partial keys."

Editing the end of a range

Topic groups See also

Call the *EditRangeEnd* procedure to put the dataset into *dsSetKey* state and start creating a list of ending values for the range. Once you call *EditRangeEnd*, subsequent assignments to the *Fields* property are treated as ending index values to use when applying the range. Fields must be indexed fields when using Paradox and dBASE.

Note: Always specify the ending values for a range, even if you want a range to end on the last record in the dataset. If you do not provide ending values, Delphi assumes the ending value of the range is a null value. A range with null ending values is always empty.

Deleting all records in a table

Topic groups See also

To delete all rows of data in a table, call a table component's *EmptyTable* method at runtime. For SQL tables, this method only succeeds if you have DELETE privilege for the table. For example, the following statement deletes all records in a dataset:

```
PhoneTable.EmptyTable;
```

Caution: Data you delete with *EmptyTable* is gone forever.

Deleting a table

Topic groups See also

At design time, to delete a table from a database, right-click the table component and select Delete Table from the context menu. The Delete Table menu pick will only be present if the table component represents an existing database table (the *DatabaseName* and *TableName* properties specify an existing table).

To delete a table at runtime, call the table component's *DeleteTable* method. For example, the following statement removes the table underlying a dataset:

```
CustomersTable.DeleteTable;
```

Caution: When you delete a table with *DeleteTable*, the table and all its data are gone forever.

Renaming a table

Topic groups See also

To rename a Paradox or dBASE table at design time, right-click the table component and select Rename Table from the context menu. You can also rename a table by typing over the name of an existing table next to the TableName property in the Object Inspector. When you change the *TableName* property, a dialog appears asking you if you want to rename the table. At this point, you can either choose to rename the table, or you can cancel the operation, changing the *TableName* property (for example, to create a new table) without changing the name of the table represented by the old value of *TableName*.

To rename a Paradox or dBASE table at runtime, call the table component's *RenameTable*method. For example, the following statement renames the Customer table to CustInfo:

```
Customer.RenameTable('CustInfo');
```

Creating a table

Topic groups See also

You can create new database tables at design time or at runtime. The Create Table command (at design time) or the *CreateTable* method (at runtime) provides a way to create tables without requiring SQL knowledge. They do, however, require you to be intimately familiar with the properties, events, and methods common to dataset components, *TTable* in particular. This is so that you can first define the table you want to create by doing the following:

- Set the *DatabaseName* property to the database that will contain the new table.
- Set the *TableType* property to the desired type of table. For Paradox, dBASE, or ASCII tables, set *TableType* to *ttParadox*, *ttDBase*, or *ttASCII*, respectively. For all other table types, set *TableType* to *ttDefault*.
- Set the *TableName* property to the name of the new table. If the value of the *TableType* property is *ttParadox* or *ttDBase*, you do not need to specify an extension.
- Add field definitions to describe the fields in the new table. At design time, you can add the field definitions by double-clicking the *FieldDefs* property in the Object Inspector to bring up the collection editor. Use the collection editor to add, remove, or change the properties of the field definitions. At runtime, clear any existing field definitions and then use the *AddFieldDef* method to add each new field definition. For each new field definition, set the properties of the *TFieldDef* object to specify the desired attributes of the field.
- Optionally, add index definitions that describe the desired indexes of the new table. At design time, you can add index definitions by double-clicking the *IndexDefs* property in the Object Inspector to bring up the collection editor. Use the collection editor to add, remove, or change the properties of the index definitions. At runtime, clear any existing index definitions, and then use the *AddIndexDef* method to add each new index definition. For each new index definition, set the properties of the *TIndexDef* object to specify the desired attributes of the index.

Note: At design time, you can preload the field definitions and index definitions of an existing table into the FieldDefs and IndexDefs properties, respectively. Set the *DatabaseName* and *TableName* properties to specify the existing table. Right click the table component and choose Update Table Definition. This automatically sets the values of the *FieldDefs* and *IndexDefs* properties to describe the fields and indexes of the existing table. Next, reset the *DatabaseName* and *TableName* to specify the table you want to create, canceling any prompts to rename the existing table. If you want to store these definitions with the table component (for example, if your application will be using them to create tables on user's systems), set the *StoreDefs* property to *True*.

Once the table is fully described, you are ready to create it. At design time, right-click the table component and choose Create Table. At runtime, call the *CreateTable* method to generate the specified table.

Warning: If you create a table that duplicates the name of an existing table, the existing table and all its data are overwritten by the newly created table. The old table and its data cannot be recovered.

The following code creates a new table at runtime and associates it with the DBDEMOS alias. Before it creates the new table, it verifies that the table name provided does not match the name of an existing table:

```
var
  NewTable: TTable;
  NewIndexOptions: TIndexOptions;
  TableFound: Boolean;
begin
  NewTable := TTable.Create;
  NewIndexOptions := [ixPrimary, ixUnique];
  with NewTable do
  begin
    Active := False;
    DatabaseName := 'DBDEMOS';
    TableName := Edit1.Text;
    TableType := ttDefault;
    FieldDefs.Clear;
```

```
    FieldDefs.Add(Edit2.Text, ftInteger, 0, False);
    FieldDefs.Add(Edit3.Text, ftInteger, 0, False);
    IndexDefs.Clear;
    IndexDefs.Add('PrimaryIndex', Edit2.Text, NewIndexOptions);
  end;
  {Now check for prior existence of this table}
  TableFound := FindTable(Edit1.Text); {code for FindTable not shown}
  if TableFound = True then
   if MessageDlg('Overwrite existing table ' + Edit1.Text + '?', mtConfirmation,
      mbYesNo, 0) = mrYes then
    TableFound := False;
  if not TableFound then
    CreateTable; { create the table}
  end;
end;
```

Importing data from another table

Topic groups See also

You can use a table component's *BatchMove*method to import data from another table. *BatchMove* can

- Copy records from another table into this table.
- Update records in this table that occur in another table.
- Append records from another table to the end of this table.
- Delete records in this table that occur in another table.

BatchMove takes two parameters: the name of the table from which to import data, and a mode specification that determines which import operation to perform. The following table describes the possible settings for the mode specification:

Value	Meaning
batAppend	Append all records from the source table to the end of this table.
batAppendUpdate	Append all records from the source table to the end of this table and update existing records in this table with the same records from the source table.
batCopy	Copy all records from the source table into this table.
batDelete	Delete all records in this table that also appear in the source table.
batUpdate	Update existing records in this table with their counterparts in the source table.

For example, the following code updates records in the current table with records from the *Customer* table:

```
Table1.BatchMove('CUSTOMER.DB', batUpdate);
```

BatchMove returns the number of records it imports successfully.

Caution: Importing records using the *batCopy* mode overwrites existing records. To preserve existing records use *batAppend* instead.

BatchMove performs only some of the functions available to your application directly through the *TBatchMove* component. If you need to move a large amount of data between or among tables, use the batch move component instead of calling a table's *BatchMove* function. For more information about using *TBatchMove*, see "Using TBatchMove."

Using TBatchMove

Topic groups See also

TBatchMove encapsulates Borland Database Engine (BDE) features that enable you to duplicate a dataset, append records from one dataset to another, update records in one dataset with records from another dataset, and delete records from one dataset that match records in another dataset. *TBatchMove* is most often used to:

- Download data from a server to a local data source for analysis or other operations.
- Move a desktop database into tables on a remote server as part of an upsizing operation.

A batch move component can create tables on the destination that correspond to the source tables, automatically mapping the column names and data types as appropriate.

The following topics are discussed in this section:

- Creating a batch move component
- Specifying a batch move mode
- Mapping data types
- Executing a batch move
- Handling batch move errors

Creating a batch move component

Topic groups See also

To create a batch move component:

1. Place the table or query component for the dataset from which you want to import records (called the *Source* dataset) on a form or in a data module.
2. Place the dataset component to which to move records (called the *Destination* dataset) on a form or in a data module.
3. Place a TBatchMove component from the Data Access page of the Component palette in a data module or on a form, and set its *Name* property to a unique value appropriate to your application.
4. Set the *Source* property of the batch move component to the name of the table to copy, append, or update from. You can select tables from the drop-down list of available dataset components.
5. Set the *Destination* property to the dataset to create, append to, or update. You can select a destination table from the drop-down list of available dataset components, or add a new table component for the destination.

   If you are appending, updating, or deleting, *Destination* must be an existing table.

   If you are copying a table and *Destination* is an existing table, then executing the batch move overwrites all of the current data in the destination table.

   If you are creating an entirely new table by copying an existing table, the resulting table has the name specified in the *Name* property of the table component to which you are copying. The resulting table type will be of a structure appropriate to the server specified by the *DatabaseName* property.
6. Set the *Mode* property to indicate the type of operation to perform. Valid operations are *batAppend* (the default), *batUpdate*, *batAppendUpdate*, *batCopy*, and *batDelete*. For more information about these modes, see "Specifying a batch move mode."
7. Optionally set the *Transliterate* property. If *Transliterate* is *True* (the default), character data is transliterated from the *Source* dataset's character set to the *Destination* dataset's character set as necessary.
8. Optionally set column mappings using the *Mappings* property. You need not set this property if you want batch move to match column based on their position in the source and destination tables. For more information about mapping columns, see "Mapping data types."
9. Optionally specify the *ChangedTableName*, *KeyViolTableName*, and *ProblemTableName* properties. Batch move stores problem records it encounters during the batch move operation in the table specified by *ProblemTableName*. If you are updating a Paradox table through a batch move, key violations can be reported in the table you specify in *KeyViolTableName*. *ChangedTableName* lists all records that changed in the destination table as a result of the batch move operation. If you do not specify these properties, these error tables are not created or used. For more information about handling batch move errors, see "Handling batch move errors."

Specifying a batch move mode

Topic groups See also

The *Mode* property specifies the operation a batch move component performs:

Property	Purpose
batAppend	Append records to the destination table.
batUpdate	Update records in the destination table with matching records from the source table. Updating is based on the current index of the destination table.
batAppendUpdate	If a matching record exists in the destination table, update it. Otherwise, append records to the destination table.
batCopy	Create the destination table based on the structure of the source table. If the destination table already exists, it is dropped and recreated.
batDelete	Delete records in the destination table that match records in the source table.

The following topics discuss these modes in more detail:

- Appending
- Updating
- Appending and updating
- Copying
- Deleting

Appending

Topic groups See also

To append data, the destination dataset must already exist. During the append operation, the BDE converts data to appropriate data types and sizes for the destination dataset if necessary. If a conversion is not possible, an exception is thrown and the data is not appended.

Updating

Topic groups See also

To update data, the destination dataset must already exist and must have an index defined that enables records to be matched. If the primary index fields are used for matching, records with index fields in the destination dataset that match index fields records in the source dataset are overwritten with the source data. During the update operation, the BDE converts data to appropriate data types and sizes for the destination dataset if necessary.

Appending and updating

Topic groups See also

To append and update data the destination dataset must already exist and must have an index defined that enables records to be matched. If the primary index fields are used for matching, records with index fields in the destination dataset that match index fields records in the source dataset are overwritten with the source data. Otherwise data from the source dataset is appended to the destination dataset. During append and update operations, the BDE converts data to appropriate data types and sizes for the destination dataset if necessary.

Copying

Topic groups See also

To make a copy of a source dataset, the destination dataset should not already exist. If it does, the batch move operation overwrites the existing destination dataset with a copy of the source dataset.

If the source and destination datasets are maintained by different types of database engines, for example, Paradox and InterBase, the BDE creates a destination dataset with a structure as close as possible to that of the source dataset and automatically performs data type and size conversions as necessary.

Note: *TBatchMove* does not copy metadata structures such as indexes, constraints, and stored procedures. You must recreate these metadata objects on your database server or through the SQL Explorer as appropriate.

Deleting

Topic groups See also

To delete data in the destination dataset, it must already exist and must have an index defined that enables records to be matched. If the primary index fields are used for matching, records with index fields in the destination dataset that match index fields records in the source dataset are deleted in the destination table.

Mapping data types

Topic groups See also

In *batAppend* mode, a batch move component creates the destination table based on the column data types of the source table. Columns and types are matched based on their position in the source and destination tables. That is, the first column in the source is matched with the first column in the destination, and so on.

To override the default column mappings, use the *Mappings* property. *Mappings* is a list of column mappings (one per line). This listing can take one of two forms. To map a column in the source table to a column of the same name in the destination table, you can use a simple listing that specifies the column name to match. For example, the following mapping specifies that a column named *ColName* in the source table should be mapped to a column of the same name in the destination table:

```
ColName
```

To map a column named *SourceColName* in the source table to a column named *DestColName* in the destination table, the syntax is as follows:

```
DestColName = SourceColName
```

If source and destination column data types are not the same, a batch move operation attempts a “best fit”. It trims character data types, if necessary, and attempts to perform a limited amount of conversion, if possible. For example, mapping a CHAR(10) column to a CHAR(5) column will result in trimming the last five characters from the source column.

As an example of conversion, if a source column of character data type is mapped to a destination of integer type, the batch move operation converts a character value of ‘5’ to the corresponding integer value. Values that cannot be converted generate errors. For more information about errors, see "Handling batch move errors.”

When moving data between different table types, a batch move component translates data types as appropriate based on the dataset’s server types. See the BDE online help file for the latest tables of mappings among server types.

Note: To batch move data to an SQL server database, you must have that database server and a version of Delphi with the appropriate SQL Link installed, or you can use ODBC if you have the proper third party ODBC drivers installed.

Executing a batch move

Topic groups See also

Use the *Execute* method to execute a previously prepared batch operation at runtime. For example, if *BatchMoveAdd* is the name of a batch move component, the following statement executes it:

```
BatchMoveAdd.Execute;
```

You can also execute a batch move at design time by right clicking the mouse on a batch move component and choosing Execute from the context menu.

The *MovedCount* property keeps track of the number of records that are moved when a batch move executes.

The *RecordCount* property specifies the maximum number of records to move. If *RecordCount* is zero, all records are moved, beginning with the first record in the source dataset. If *RecordCount* is a positive number, a maximum of *RecordCount* records are moved, beginning with the current record in the source dataset. If *RecordCount* is greater than the number of records between the current record in the source dataset and its last record, the batch move terminates when the end of the source dataset is reached. You can examine *MoveCount* to determine how many records were actually transferred.

Handling batch move errors

Topic groups See also

There are two types of errors that can occur in a batch move operation: data type conversion errors and integrity violations. *TBatchMove* has a number of properties that report on and control error handling.

The *AbortOnProblem* property specifies whether to abort the operation when a data type conversion error occurs. If *AbortOnProblem* is *True*, the batch move operation is canceled when an error occurs. If *False*, the operation continues. You can examine the table you specify in the *ProblemTableName* to determine which records caused problems.

The *AbortOnKeyViol* property indicates whether to abort the operation when a Paradox key violation occurs.

The *ProblemCount* property indicates the number of records that could not be handled in the destination table without a loss of data. If *AbortOnProblem* is *True*, this number is one, since the operation is aborted when an error occurs.

The following properties enable a batch move component to create additional tables that document the batch move operation:

- *ChangedTableName*, if specified, creates a local Paradox table containing all records in the destination table that changed as a result of an update or delete operation.
- *KeyViolTableName,* if specified, creates a local Paradox table containing all records from the source table that caused a key violation when working with a Paradox table. If *AbortOnKeyViol* is *True*, this table will contain at most one entry since the operation is aborted on the first problem encountered.
- *ProblemTableName,* if specified, creates a local Paradox table containing all records that could not be posted in the destination table due to data type conversion errors. For example, the table could contain records from the source table whose data had to be trimmed to fit in the destination table. If *AbortOnProblem* is *True*, there is at most one record in this table since the operation is aborted on the first problem encountered.

Note: If *ProblemTableName* is not specified, the data in the record is trimmed and placed in the destination table.

Synchronizing tables linked to the same database table

Topic groups See also

If more than one table component is linked to the same database table through their *DatabaseName* and *TableName* properties and the tables do not share a data source component, then each table has its own view on the data and its own current record. As users access records through each table component, the components' current records will differ.

You can force the current record for each of these table components to be the same with the *GotoCurrent* method. *GotoCurrent* sets its own table's current record to the current record of another table component. For example, the following code sets the current record of *CustomerTableOne* to be the same as the current record of *CustomerTableTwo*:

```
CustomerTableOne.GotoCurrent(CustomerTableTwo);
```

Tip: If your application needs to synchronize table components in this manner, put the components in a data module and include the header for the data module in each unit that accesses the tables.

If you must synchronize table components on separate forms, you must include one form's header file in the source unit of the other form, and you must qualify at least one of the table names with its form name.

For example:

```
CustomerTableOne.GotoCurrent(Form2.CustomerTableTwo);
```

Creating master/detail forms

Topic groups See also

A table component's *MasterSource* and *MasterFields* properties can be used to establish 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 data source component in this property.

The *MasterFields* property specifies the column(s) common to both tables used to establish the link. To link tables based on multiple column names, use a semicolon delimited list:

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

To help create meaningful links between two tables, you can use the Field Link designer. For more information about the Field Link designer, see the User's Guide.

For an example of a master/detail form, see "Building an example master/detail form."

Building an example master/detail form

Topic groups

The following steps create a simple form in which a user can scroll through customer records and display all orders for the current customer. The master table is the *CustomersTable* table, and the detail table is *OrdersTable*.

1. Place two *TTable* and two *TDataSource* components in a data module.
2. Set the properties of the first *TTable* component as follows:
   - *DatabaseName*: DBDEMOS
   - *TableName*: CUSTOMER
   - *Name*: CustomersTable
3. Set the properties of the second *TTable* component as follows:
   - *DatabaseName*: DBDEMOS
   - *TableName*: ORDERS
   - *Name*: OrdersTable
4. Set the properties of the first *TDataSource* component as follows:
   - *Name*: CustSource
   - *DataSet*: CustomersTable
5. Set the properties of the second *TDataSource* component as follows:
   - *Name*: OrdersSource
   - *DataSet*: OrdersTable
6. Place two *TDBGrid* components on a form.
7. Choose File|Include Unit Hdr to specify that the form should use the data module.
8. Set the *DataSource* property of the first grid component to "DataModule2->CustSource", and set the *DataSource* property of the second grid to "DataModule2->OrdersSource".
9. Set the *MasterSource* property of *OrdersTable* to "CustSource". This links the CUSTOMER table (the master table) to the ORDERS table (the detail table).
10. Double-click the *MasterFields* property value box in the Object Inspector to invoke the Field Link Designer to set the following properties:
    - In the Available Indexes field, choose *CustNo* to link the two tables by the *CustNo* field.
    - Select *CustNo* in both the Detail Fields and Master Fields field lists.
    - Click the Add button to add this join condition. In the Joined Fields list, "CustNo -> CustNo" appears.
    - Choose OK to commit your selections and exit the Field Link Designer.
11. Set the *Active* properties of *CustomersTable* and *OrdersTable* to *True* to display data in the grids on the form.
12. Compile and run the application.

If you run the application now, you will see that the tables are linked together, and that when you move to a new record in the CUSTOMER table, you see only those records in the ORDERS table that belong to the current customer.

Working with nested tables

Topic groups See also

A nested table component provides access to data in a nested dataset of a table. The NestedDataSet property of a persistent nested dataset field contains a reference to the nested dataset. Since *TNestedDataSet* descends from *TBDEDataSet*, a nested table inherits BDE functionality, and so uses the Borland Database Engine (BDE) to access the nested table data. A nested table provides much of the functionality of a table component, but the data it accesses is stored in a nested table.

Setting up a nested table component

The following steps are general instructions for setting up a nested table component at design time. A table or live query component must be available that accesses a dataset containing a dataset or reference field. A persistent field for the *TDataSetField* or *TReferenceField* must also already exist. See Working with dataset fields.

To use a nested table component,

1. Place a nested table component from the Data Access page of the Component palette in a data module or on a form, and set its *Name* property to a unique value appropriate to your application.
2. Set the *DataSetField* of the component to the name of the persistent dataset field or reference field to access. You can select fields from the drop-down list.
3. Place a data source component in the data module or on the form, and set its *DataSet* property to the name of the nested table component. The data source component is used to pass a result set from the table to data-aware components for display.

Working with queries

Topic groups

This section describes the *TQuery* dataset component which enables you to use SQL statements to access data. It assumes you are familiar with the general discussion of datasets and data sources in Understanding datasets.

A query component encapsulates an SQL statement that is used in a client application to retrieve, insert, update, and delete data from one or more database tables. SQL is an industry-standard relational database language that is used by most remote, server-based databases, such as Sybase, Oracle, InterBase, and Microsoft SQL Server. Query components can be used with remote database servers (if your version of Delphi includes SQL links), with Paradox, dBASE, FoxPro, and Access, and with ODBC-compliant databases.

The following topics are discussed in this section:

- Using queries effectively
- What databases can you access with a query component?
- Using a query component: an overview
- Specifying the SQL statement to execute
- Setting parameters
- Executing a query
- Preparing a query
- Unpreparing a query to release resources
- Creating heterogeneous queries
- Improving query performance
- Working with result sets

Using queries effectively

Topic groups See also

To use the query component effectively, you must be familiar with

- SQL and your server's SQL implementation, including limitations and extensions to the SQL-92 standard
- Borland Database Engine (BDE)

If you are an experienced desktop database developer moving to server-based applications, see Queries for desktop developers for an introduction to queries before reading the remainder of this section. If you are new to SQL, you may want to purchase one of the many fine third party books that cover SQL in-depth. One of the best is *Understanding the New SQL: A Complete Guide*, by Jim Melton and Alan R. Simpson, Morgan Kaufmann Publishers.

If you are an experienced database server developer, but are new to building Delphi clients, then you are already familiar with SQL and your server, but you may be unfamiliar with the BDE. See Queries for server developers for an introduction to queries and the BDE before reading the remainder of this section.

The following topics are discussed in this section:

- Queries for desktop developers
- Queries for server developers

Queries for desktop developers

Topic groups See also

As a desktop developer you are already familiar with the basic table, record, and field paradigm used by Delphi and the BDE. You feel very comfortable using a *TTable* component to gain access to every field in every data record in a dataset. You know that when you set a table's *TableName* property, you specify the database table to access.

Chances are you have also used a *TTable*'s range methods and filter property to limit the number of records available at any given time in your applications. Applying a range temporarily limits data access to a block of contiguously indexed records that fall within prescribed boundary conditions, such as returning all records for employees whose last names are greater than or equal to "Jones" and less than or equal to "Smith". Setting a filter temporarily restricts data access to a set of records that is usually noncontiguous and that meets filter criteria, such as returning only those customer records that have a California mailing address.

A query behaves in many ways very much like a table filter, except that you use the query component's SQL property (and sometimes the *Params* property) to identify the records in a dataset to retrieve, insert, delete, or update. In some ways a query is even more powerful than a filter because it lets you access

- More than one table at a time (called a "join" in SQL).
- A specified subset of rows *and* columns in its underlying table(s), rather than always returning all rows and columns. This improves both performance and security. Memory is not wasted on unnecessary data, and you can prevent access to fields a user should not view or modify.

Queries can be verbatim, or they can contain replaceable parameters. Queries that use parameters are called *parameterized queries*. When you use parameterized queries, the actual values assigned to the parameters are inserted into the query by the BDE before you execute, or run, the query. Using parameterized queries is very flexible, because you can change a user's view of and access to data on the fly at runtime without having to alter the SQL statement.

Most often you use queries to select the data that a user should see in your application, just as you do when you use a table component. Queries, however, can also perform update, insert, and delete operations instead of retrieving records for display. When you use a query to perform insert, update, and delete operations, the query ordinarily does not return records for viewing. In this way a query differs from a table.

To learn more about using the SQL property to write an SQL statement, see Specifying the SQL statement to execute.. To learn more about using parameters in your SQL statements, see Setting parameters.. To learn about executing a query, see Executing a query..

Queries for server developers

Topic groups See also

As a server developer you are already familiar with SQL and with the capabilities of your database server. To you a query is the SQL statement you use to access data. You know how to use and manipulate this statement and how to use optional parameters with it.

The SQL statement and its parameters are the most important parts of a query component. The query component's SQL property is used to provide the SQL statement to use for data access, and the component's *Params* property is an optional array of parameters to bind into the query. However, a query component is much more than an SQL statement and its parameters. A query component is also the interface between your client application and the BDE.

A client application uses the properties and methods of a query component to manipulate an SQL statement and its parameters, to specify the database to query, to prepare and unprepare queries with parameters, and to execute the query. A query component's methods call the BDE, which, in turn, processes your query requests, and communicates with the database server, usually through an SQL Links driver. The server passes a result set, if appropriate, back to the BDE, and the BDE returns it to your application through the query component.

When you work with a query component, you should be aware that some of the terminology used to describe BDE features can be confusing at first because it has very different meanings than what you are familiar with as an SQL database programmer. For example, the BDE commonly uses the term "alias" to refer to a shorthand name for the path to the database server. The BDE alias is stored in a configuration file, and is set in the query component's *DatabaseName* property. (You can still use table aliases, or "table correlation names," in your SQL statements.)

Similarly, the BDE help documentation, available online in \Program Files\Common Files\Borland Shared\BDE\BDE32.HLP, often refers to queries that use parameters as "parameterized queries" where you are more likely to think of SQL statements that use bound variables or parameter bindings.

Note: This section uses BDE terminology because you will encounter it throughout our documentation. Whenever confusion may result from using BDE terms, explanations are provided.

To learn more about using the SQL property to write an SQL statement, see Specifying the SQL statement to execute.. To learn more about using parameters in your SQL statements, see Setting parameters.. To learn about preparing a query, see Preparing a query., and to learn more about executing a query, see Executing a query..

What databases can you access with a query component?

Topic groups See also

A TQuery component can access data in:

- Paradox or dBASE tables, using Local SQL, which is part of the BDE. Local SQL is a subset of the SQL-92 specification. Most DML is supported and enough DDL syntax to work with these types of tables. See the local SQL help, LOCALSQL.HLP, for details on supported SQL syntax.
- Local InterBase Server databases, using the InterBase engine. For information on InterBase's SQL-92 standard SQL syntax support and extended syntax support, see the InterBase *Language Reference.*
- Databases on remote database servers such as Oracle, Sybase, MS-SQL Server, Informix, DB2, and InterBase (depending on your version of Delphi). You must have installed the appropriate SQL Link driver and client software (vendor-supplied) specific to the database server to access a remote server. Any standard SQL syntax supported by these servers is allowed. For information on SQL syntax, limitations, and extensions, see the documentation for your particular server.

Delphi also supports heterogeneous queries against more than one server or table type (for example, data from an Oracle table and a Paradox table). When you create a heterogeneous query, the BDE uses Local SQL to process the query. For more information, see Creating heterogeneous queries..

Using a query component: an overview

Topic groups See also

To use a query component in an application, follow these steps at design time:

1. Place a query component from the Data Access tab of the Component palette in a data module, and set its *Name* property appropriately for your application.
2. Set the *DatabaseName* property of the component to the name of the database to query. *DatabaseName* can be a BDE alias, an explicit directory path (for local tables), or the value from the *DatabaseName* property of a *TDatabase* component in the application.
3. Specify an SQL statement in the SQL property of the component, and optionally specify any parameters for the statement in the *Params* property. For more information, see Specifying the SQL property at design time..
4. If the query data is to be used with visual data controls, place a data source component from the Data Access tab of the Component palette in the data module, and set its *DataSet* property to the name of the query component. The data source component is used to return the results of the query (called a *result set*) from the query to data-aware components for display. Connect data-aware components to the data source using their *DataSource* and *DataField* properties.
5. Activate the query component. For queries that return a result set, use the *Active* property or the *Open* method. For queries that only perform an action on a table and return no result set, use the *ExecSQL* method.

To execute a query for the first time at runtime, follow these steps:

1. Close the query component.
2. Provide an SQL statement in the SQL property if you did not set the SQL property at design time, or if you want to change the SQL statement already provided. To use the design-time statement as is, skip this step. For more information about setting the SQL property, see Specifying the SQL statement to execute..
3. Set parameters and parameter values in the *Params* property either directly or by using the *ParamByName* method. If a query does not contain parameters, or the parameters set at design time are unchanged, skip this step. For more information about setting parameters, see Setting parameters..
4. Call *Prepare* to initialize the BDE and bind parameter values into the query. Calling *Prepare* is optional, though highly recommended. For more information about preparing a query, see Preparing a query.
5. Call *Open* for queries that return a result set, or call *ExecSQL* for queries that do not return a result set. For more information about opening and executing a query see Executing a query.

After you execute a query for the first time, then as long as you do not modify the SQL statement, an application can repeatedly close and reopen or re-execute a query without preparing it again. For more information about reusing a query, see Executing a query.

Specifying the SQL statement to execute

Topic groups See also

Use the SQL property to specify the SQL query statement to execute. At design time a query is prepared and executed automatically when you set the query component's *Active* property to *true*. At runtime, a query is prepared with a call to *Prepare*, and executed when the application calls the component's *Open* or *ExecSQL* methods.

The SQL property is a *TStrings* object, which is an array of text strings and a set of properties, events, and methods that manipulate them. The strings in SQL are automatically concatenated to produce the SQL statement to execute. You can provide a statement in as few or as many separate strings as you desire. One advantage to using a series of strings is that you can divide the SQL statement into logical units (for example, putting the WHERE clause for a SELECT statement into its own string), so that it is easier to modify and debug a query.

The SQL statement can be a query that contains hard-coded field names and values, or it can be a parameterized query that contains replaceable parameters that represent field values that must be bound into the statement before it is executed. For example, this statement is hard-coded:

```
SELECT * FROM Customer WHERE CustNo = 1231
```

Hard-coded statements are useful when applications execute exact, known queries each time they run. At design time or runtime you can easily replace one hard-code query with another hard-coded or parameterized query as needed. Whenever the SQL property is changed the query is automatically closed and unprepared.

Note: In queries made against the BDE engine using local SQL, when column names in a query contain spaces or special characters, the column name must be enclosed in quotes and must be preceded by a table reference and a period. For example, BIOLIFE."Species Name". For information about valid column names, see your server's documentation.

A parameterized query contains one or more placeholder parameters, application variables that stand in for comparison values such as those found in the WHERE clause of a SELECT statement. Using parameterized queries enables you to change the value without rewriting the application. Parameter values must be bound into the SQL statement before it is executed for the first time. Query components do this automatically for you even if you do not explicitly call the *Prepare* method before executing a query.

This statement is a parameterized query:

```
SELECT * FROM Customer WHERE CustNo = :Number
```

The variable *Number*, indicated by the leading colon, is a parameter that fills in for a comparison value that must be provided at runtime and that may vary each time the statement is executed. The actual value for *Number* is provided in the query component's *Params* property.

Tip: It is a good programming practice to provide variable names for parameters that correspond to the actual name of the column with which it is associated. For example, if a column name is "Number," then its corresponding parameter would be ":Number". Using matching names ensures that if a query uses its *DataSource* property to provide values for parameters, it can match the variable name to valid field names.

The following topics are discussed in this section:

- Specifying the SQL property at design time
- Specifying an SQL statement at runtime

Specifying the SQL property at design time

Topic groups See also

You can specify the SQL property at design time using the String List editor. To invoke the String List editor for the SQL property:

- Double-click on the SQL property value column, or
- Click its ellipsis button.

You can enter an SQL statement in as many or as few lines as you want. Entering a statement on multiple lines, however, makes it easier to read, change, and debug. Choose OK to assign the text you enter to the SQL property.

Normally, the SQL property can contain only one complete SQL statement at a time, although these statements can be as complex as necessary (for example, a SELECT statement with a WHERE clause that uses several nested logical operators such as AND and OR). Some servers also support “batch” syntax that permits multiple statements; if your server supports such syntax, you can enter multiple statements in the SQL property.

Note: With some versions of Delphi, you can also use the SQL Builder to construct a query based on a visible representation of tables and fields in a database. To use the SQL Builder, select a query component, right-click it to invoke the context menu, and choose Graphical Query Editor. To learn how to use the SQL Builder, open it and use its online help.

Specifying an SQL statement at run time

Topic groups See also

There are three ways to set the SQL property at runtime. An application can set the SQL property directly, it can call the SQL property's *LoadFromFile* method to read an SQL statement from a file, or an SQL statement in a string list object can be assigned to the SQL property.

- Setting the SQL property directly
- Loading the SQL property from a file
- Loading the SQL property from string list object

Setting the SQL property directly

Topic groups See also

To directly set the SQL property at runtime,

1. Call *Close* to deactivate the query. Even though an attempt to modify the SQL property automatically deactivates the query, it is a good safety measure to do so explicitly.
2. If you are replacing the whole SQL statement, call the *Clear* method for the SQL property to delete its current SQL statement.
3. If you are building the whole SQL statement from nothing or adding a line to an existing statement, call the *Add* method for the SQL property to insert and append one or more strings to the SQL property to create a new SQL statement. If you are modifying an existing line use the SQL property with an index to indicate the line affected, and assign the new value.
4. Call *Open* or *ExecSQL* to execute the query.

The following code illustrates building an entire SQL statement from nothing.

```
with CustomerQuery do begin
  Close;                                              { close the query if it's
active }
  with SQL do begin
    Clear;                                  { delete the current SQL statement, if
any }
    Add('SELECT * FROM Customer');                          { add first line of
SQL... }
    Add('WHERE Company = "Sight Diver"');                    { ... and second
line }
  end;
  Open;                                                      { activate the
query }
end;
```

The code below demonstrates modifying only a single line in an existing SQL statement. In this case, the WHERE clause already exists on the second line of the statement. It is referenced via the SQL property using an index of 1.

```
CustomerQuery.SQL[1] := 'WHERE Company = "Kauai Dive Shoppe"';
```

Note: If a query uses parameters, you should also set their initial values and call the *Prepare* method before opening or executing a query. Explicitly calling *Prepare* is most useful if the same SQL statement is used repeatedly; otherwise it is called automatically by the query component.

Loading the SQL property from a file

Topic groups See also

You can also use the *LoadFromFile* method to assign an SQL statement in a text file to the SQL property. The *LoadFromFile* method automatically clears the current contents of the SQL property before loading the new statement from file. For example:

```
CustomerQuery.Close;
CustomerQuery.SQL.LoadFromFile('c:\orders.txt');
CustomerQuery.Open;
```

Note: If the SQL statement contained in the file is a parameterized query, set the initial values for the parameters and call *Prepare* before opening or executing the query. Explicitly calling *Prepare* is most useful if the same SQL statement is used repeatedly; otherwise it is called automatically by the query component.

Loading the SQL property from string list object

Topic groups See also

You can also use the *Assign* method of the SQL property to copy the contents of a string list object into the SQL property. The *Assign* method automatically clears the current contents of the SQL property before copying the new statement. For example, copying an SQL statement from a *TMemo* component:

```
CustomerQuery.Close;
CustomerQuery.SQL.Assign(Memo1.Lines);
CustomerQuery.Open;
```

Note: If the SQL statement is a parameterized query, set the initial values for the parameters and call *Prepare* before opening or executing the query. Explicitly calling *Prepare* is most useful if the same SQL statement is used repeatedly; otherwise it is called automatically by the query component.

Setting parameters

Topic groups See also

A parameterized SQL statement contains parameters, or variables, the values of which can be varied at design time or runtime. Parameters can replace data values, such as those used in a WHERE clause for comparisons, that appear in an SQL statement. Ordinarily, parameters stand in for data values passed to the statement. For example, in the following INSERT statement, values to insert are passed as parameters:

```
INSERT INTO Country (Name, Capital, Population)
VALUES (:Name, :Capital, :Population)
```

In this SQL statement, *:name*, *:capital*, and *:population* are placeholders for actual values supplied to the statement at runtime by your application. Before a parameterized query is executed for the first time, your application should call the *Prepare* method to bind the current values for the parameters to the SQL statement. Binding means that the BDE and the server allocate resources for the statement and its parameters that improve the execution speed of the query.

```
with Query1 do begin
  Close;
  Unprepare;
  ParamByName('Name').AsString := 'Belize';
  ParamByName('Capital').AsString := 'Belmopan';
  ParamByName('Population').AsInteger := '240000';
  Prepare;
  Open;
end;
```

The following topics are discussed in this section:

- Supplying parameters at design time
- Supplying parameters at runtime
- Using a data source to bind parameters

Supplying parameters at design time

Topic groups See also

At design time, parameters in the SQL statement appear in the parameter collection editor. To access the *TParam* objects for the parameters, invoke the parameter collection editor, select a parameter, and access the *TParam* properties in the Object Inspector. If the SQL statement does not contain any parameters, no TParam objects are listed in the collection editor. You can only add parameters by writing them in the SQL statement.

To access parameters,

1. Select the query component.
2. Click on the ellipsis button for the *Params* property in Object Inspector.
3. In the parameter collection editor, select a parameter.
4. The *TParam* object for the selected parameter appears in the Object Inspector.
5. Inspect and modify the properties for the *TParam* in the Object Inspector.

For queries that do not already contain parameters (the SQL property is empty or the existing SQL statement has no parameters), the list of parameters in the collection editor dialog is empty. If parameters are already defined for a query, then the parameter editor lists all existing parameters.

Note: The *TQuery* component shares the *TParam* object and its collection editor with a number of different components. While the right-click context menu of the collection editor always contains the Add and Delete options, they are never enabled for *TQuery* parameters. The only way to add or delete *TQuery* parameters is in the SQL statement itself.

As each parameter in the collection editor is selected, the Object Inspector displays the properties and events for that parameter. Set the values for parameter properties and methods in the Object Inspector.

The *DataType* property lists the BDE data type for the parameter selected in the editing dialog. Initially the type will be *ftUnknown*. You must set a data type for each parameter. In general, BDE data types conform to server data types. For specific BDE-to-server data type mappings, see the BDE help in \Program Files\Common Files\Borland Shared\BDE\BDE32.HLP.

The *ParamType* property lists the type of parameter selected in the editing dialog. Initially the type will be *ptUnknown*. You must set a type for each parameter.

Use the *Value* property to specify a value for the selected parameter at design-time. This is not mandatory when parameter values are supplied at runtime. In these cases, lease *Value* blank.

Supplying parameters at run time

Topic groups See also

To create parameters at runtime, you can use the

- *ParamByName* method to assign values to a parameter based on its name.
- *Params* property to assign values to a parameter based on the parameter's ordinal position within the SQL statement.
- *Params.ParamValues* property to assign values to one or more parameters in a single command line, based on the name of each parameter set. This method uses variants and avoids the need to cast values.

For all of the examples below, assume the SQL property contains the SQL statement below. All three parameters used are of data type *ftString*.

```
INSERT INTO "COUNTRY.DB"
(Name, Capital, Continent)
VALUES (:Name, :Capital, :Continent)
```

The following code uses *ParamByName* to assign the text of an edit box to the Capital parameter:

```
Query1.ParamByName('Capital').AsString := Edit1.Text;
```

The same code can be rewritten using the *Params* property, using an index of 1 (the Capital parameter is the second parameter in the SQL statement):

```
Query1.Params[1].AsString := Edit1.Text;
```

The command line below sets all three parameters at once, using the *Params.ParamValues* property:

```
Query1.Params.ParamValues['Country;Capital;Continent'] :=
  VarArrayOf([Edit1.Text, Edit2.Text, Edit3.Text]);
```

Using a data source to bind parameters

Topic groups See also

If parameter values for a parameterized query are not bound at design time or specified at runtime, the query component attempts to supply values for them based on its *DataSource* property. *DataSource* specifies a different table or query component that the query component can search for field names that match the names of unbound parameters. This search dataset must be created and populated before you create the query component that uses it. If matches are found in the search dataset, the query component binds the parameter values to the values of the fields in the current record pointed to by the data source.

You can create a simple application to understand how to use the *DataSource* property to link a query in a master-detail form. Suppose the data module for this application is called, *LinkModule*, and that it contains a query component called *OrdersQuery* that has the following SQL property:

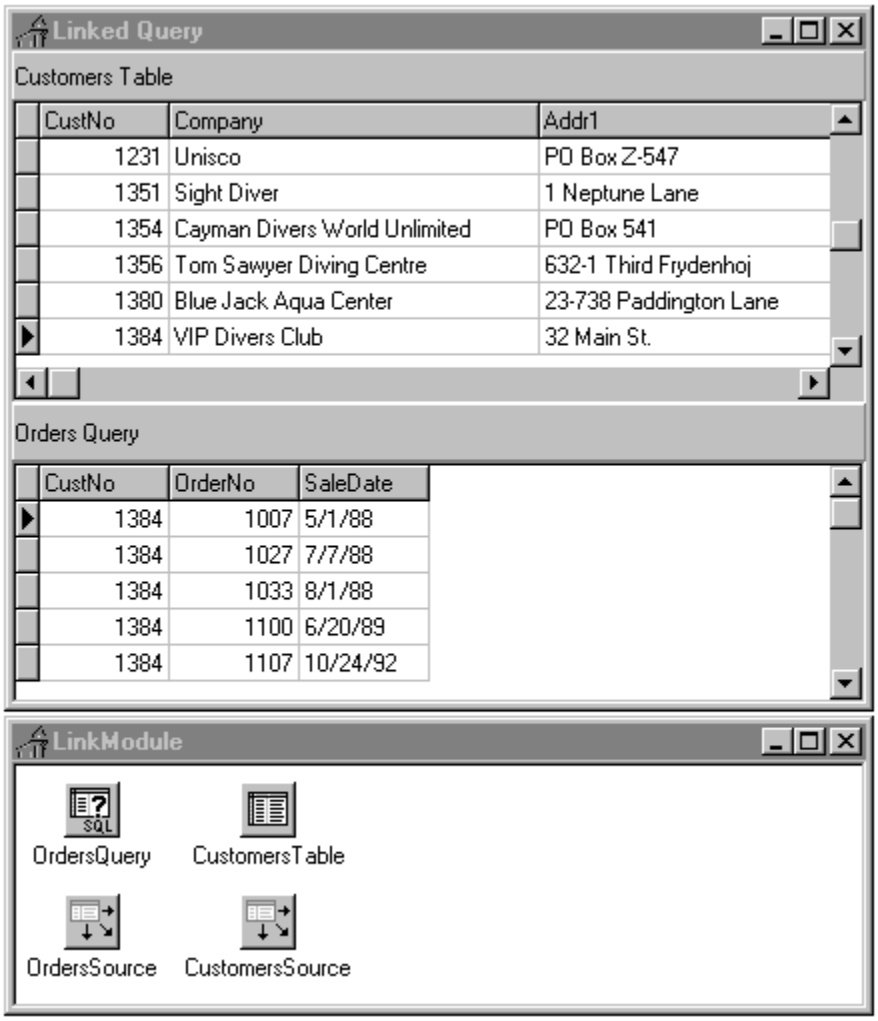
```
SELECT CustNo, OrderNo, SaleDate
FROM Orders
WHERE CustNo = :CustNo
```

The *LinkModule* data module also contains:

- A *TTable* dataset component named *CustomersTable* linked to the CUSTOMER.DB table.
- A *TDataSource* component named *OrdersSource*. The *DataSet* property of *OrdersSource* points to *OrdersQuery*.
- A *TDataSource* named *CustomersSource* linked to *CustomersTable*. The *DataSource* property of the *OrdersQuery* component is also set to *CustomersSource*. This is the setting that makes *OrdersQuery* a linked query.

Suppose, too, that this application has a form, named *LinkedQuery* that contains two data grids, a *Customers Table* grid linked to *CustomersSource*, and an *Order Quer*y grid linked to *OrdersSource*.

The following figure illustrates how this application appears at design time.



Note: If you build this application, create the table component and its data source before creating the query component.

If you compile this application, at runtime the *:CustNo* parameter in the SQL statement for *OrdersQuery* is not assigned a value, so *OrdersQuery* tries to match the parameter by name against a column in the table pointed to by *CustomersSource*. *CustomersSource* gets its data from *CustomersTable*, which, in turn, derives its data from the CUSTOMER.DB table. Because CUSTOMER.DB contains a column called “CustNo,” the value from the *CustNo* field in the current record of the *CustomersTable* dataset is assigned to the *:CustNo* parameter for the *OrdersQuery* SQL statement. The grids are linked in a master-detail relationship. At runtime, each time you select a different record in the Customers Table grid, the *OrdersQuery* SELECT statement executes to retrieve all orders based on the current customer number.

Executing a query

Topic groups See also

After you specify an SQL statement in the SQL property and set any parameters for the query, you can execute the query. When a query is executed, the BDE receives and processes SQL statements from your application. If the query is against local tables (dBASE and Paradox), the BDE SQL engine processes the SQL statement and, for a SELECT query, returns data to the application. If the query is against an SQL server table and the *TQuery.RequestLive* property is set to *False*, the SQL statement is not processed or interpreted by the BDE, but is passed directly to the database system. If the *RequestLive* property is set to *True*, the SQL statement will need to abide by local SQL standards as the BDE will need to use it for conveying data changes to the table.

Note: Before you execute a query for the first time, you may want to call the *Prepare* method to improve query performance. *Prepare* initializes the BDE and the database server, each of which allocates system resources for the query. For more information about preparing a query, see "Preparing a query.".

The following sections describe executing both static and dynamic SQL statements at design time and at runtime.

The following topics are discussed in this section:

- Executing a query at design time
- Executing a query at runtime

Executing a query at design time

Topic groups See also

To execute a query at design time, set its Active property to *true* in the Object Inspector.

The results of the query, if any, are displayed in any data-aware controls associated with the query component.

Note: The *Active* property can be used only with queries that returns a result set, such as by the SELECT statement.

Executing a query at runtime

Topic groups See also

To execute a query at runtime, use one of the following methods:

- *Open* executes a query that returns a result set, such as with the SELECT statement.
- *ExecSQL* executes a query that does not return a result set, such as with the INSERT, UPDATE, or DELETE statements.

Note: If you do not know at design time whether a query will return a result set at runtime, code both types of query execution statements in a **try..except** block. Put a call to the *Open* method in the **try** clause. This allows you to suppress the error message that would occur due to using an activate method not applicable to the type of SQL statement used. Check the type of exception that occurs. If it is other than an *ENoResult* exception, the exception occurred for another reason and must be processed. This works because an action query will be executed when the query is activated with the *Open* method, but an exception occurs in addition to that.

```
try
  Query2.Open;
except
  on E: Exception do
    if not (E is ENoResultSet) then
      raise;
end;
```

The following topics are discussed in this section:

- Executing a query that returns a result set
- Executing a query without a result set

Executing a query that returns a result set

Topic groups See also

To execute a query that returns a result set (a query that uses a SELECT statement), follow these steps:

1. Call *Close* to ensure that the query is not already open. If a query is already open you cannot open it again without first closing it. Closing a query and reopening it fetches a new version of data from the server.
2. Call *Open* to execute the query.

For example:

```
CustomerQuery.Close;
CustomerQuery.Open; { query returns a result set }
```

For information on navigating within a result set, see Disabling bi-directional cursors.. For information on editing and updating a result set, see Working with result sets.

Executing a query without a result set

Topic groups See also

To execute a query that does not return a result set (a query that has an SQL statement such as INSERT, UPDATE, or DELETE), call *ExecSQL* to execute the query.

For example:

```
CustomerQuery.ExecSQL;  { query does not return a result set }
```

Preparing a query

Topic groups See also

Preparing a query is an optional step that precedes query execution. Preparing a query submits the SQL statement and its parameters, if any, to the BDE for parsing, resource allocation, and optimization. The BDE, in turn, notifies the database server to prepare for the query. The server, too, may allocate resources for the query. These operations improve query performance, making your application faster, especially when working with updatable queries.

An application can prepare a query by calling the *Prepare* method. If you do not prepare a query before executing it, then Delphi automatically prepares it for you each time you call *Open* or *ExecSQL*. Even though Delphi prepares queries for you, it is better programming practice to prepare a query explicitly. That way your code is self-documenting, and your intentions are clear. For example:

```
CustomerQuery.Close;
if not (CustomerQuery.Prepared) then
  CustomerQuery.Prepare;
CustomerQuery.Open;
```

This example checks the query component's *Prepared* property to determine if a query is already prepared. *Prepared* is a Boolean value that is *true* if a query is already prepared. If the query is not already prepared, the example calls the *Prepare* method before calling *Open*.

Unpreparing a query to release resources

Topic groups See also

The *UnPrepare* method sets the *Prepared* property to *false*. *UnPrepare*

- Ensures that the SQL property is prepared prior to executing it again.
- Notifies the BDE to release the internal resources allocated for the statement.
- Notifies the server to release any resources it has allocated for the statement.

To unprepare a query, call

```
CustomerQuery.UnPrepare;
```

Note: When you change the text of the SQL property for a query, the query component automatically closes and unprepares the query.

Creating heterogenous queries

Topic groups See also

Delphi supports *heterogeneous queries*, that is, queries made against tables in more than one database. A heterogeneous 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. When you execute a heterogeneous query, the BDE parses and processes the query using Local SQL. Because BDE uses Local SQL, extended, server-specific SQL syntax is not supported.

To perform a heterogeneous query, follow these steps:

1. Define separate BDE aliases for each database accessed in the query. Leave the *DatabaseName* property of the *TQuery* blank; the names of the two databases used will be specified in the SQL statement.
2. Specify the SQL statement to execute in the SQL property. Precede each table name in the SQL statement with the BDE alias for the database where that table can be found. The table reference is preceded by the name of the BDE alias, enclosed in colons. This whole reference is then enclosed in quotation marks.
3. Set any parameters for the query in the *Params* property.
4. Call *Prepare* to prepare the query for execution prior to executing it for the first time.
5. Call *Open* or *ExecSQL* depending on the type of query to execute.

For example, suppose you define an alias called *Oracle1* for an Oracle database that has a CUSTOMER table, and *Sybase1* for a Sybase database that has an ORDERS table. A simple query against these two tables would be:

```
SELECT Customer.CustNo, Orders.OrderNo
FROM ":Oracle1:CUSTOMER"
  JOIN ":Sybase1:ORDERS"
    ON (Customer.CustNo = Orders.CustNo)
WHERE (Customer.CustNo = 1503)
```

As an alternative to using a BDE alias to specify the database in a heterogeneous query, you can use a *TDatabase* component. Configure the *TDatabase* as normal to point to the database, set the *TDatabase.DatabaseName* to an arbitrary but unique value, and then use that value in the SQL statement instead of a BDE alias name.

Improving query performance

Topic groups See also

Following are steps you can take to improve query execution speed:

- Set a query's *UniDirectional* property to *true* if you do not need to navigate backward through a result set (SQL-92 does not, itself, permit backward navigation through a result set). By default, *UniDirectional* is *false* because the BDE supports bi-directional cursors by default.
- Prepare the query before execution. This is especially helpful when you plan to execute a single query several times. You need only prepare the query once, before its first use. For more information about query preparation, see Preparing a query.

For information on using the *UniDirectional* property, see Disabling bi-directional cursors.

Disabling bi-directional cursors

Topic groups

The *UniDirectional* property determines whether or not BDE bi-directional cursors are enabled for a query. When a query returns a result set, it also receives a cursor, or pointer to the first record in that result set. The record pointed to by the cursor is the currently active record. The current record is the one whose field values are displayed in data-aware components associated with the result set's data source.

UniDirectional is *false* by default, meaning that the cursor for a result set can navigate both forward and backward through its records. Bi-directional cursor support requires some additional processing overhead, and can slow some queries. To improve query performance, you may be able to set *UniDirectional* to *true*, restricting a cursor to forward movement through a result set.

If you do not need to be able to navigate backward through a result set, you can set *UniDirectional* to *true* for a query. Set *UniDirectional* before preparing and executing a query. The following code illustrates setting *UniDirectional* prior to preparing and executing a query:

```
if not (CustomerQuery.Prepared) then begin
  CustomerQuery.UniDirectional := True;
  CustomerQuery.Prepare;
end;
CustomerQuery.Open;  { returns a result set with a one-way cursor }
```

Working with result sets

Topic groups See also

By default, the result set returned by a query is read-only. Your application can display field values from the result set in data-aware controls, but users cannot edit those values. To enable editing of a result set, your application must request a “live” result set.

The following topics are discussed in this section:

- Enabling editing of a result set
- Local SQL requirements for a live result set
- Remote server SQL requirements for a live result set
- Restrictions on updating a live result set
- Updating a read-only result set

Enabling editing of a result set

Topic groups See also

To request a result set that users can edit in data-aware controls, set a query component's *RequestLive* property to *true*. Setting *RequestLive* to *true* does not guarantee a live result set, but the BDE attempts to honor the request whenever possible. There are some restrictions on live result set requests, depending on whether or not a query uses the local SQL parser or a server's SQL parser. Heterogeneous joins and queries executed against Paradox or dBASE are parsed by the BDE using local SQL. Queries against a remote database server are parsed by the server.

If an application requests and receives a live result set, the *CanModify* property of the query component is set to *true*.

If an application requests a live result set, but the SELECT statement syntax does not allow it, the BDE returns either

- A read-only result set for queries made against Paradox or dBASE.
- An error code for pass-through SQL queries made against a remote server.

Local SQL requirements for a live result set

Topic groups See also

For queries that use the local SQL parser, the BDE offers expanded support for updatable, live result sets in both single table and multi-table queries. The local SQL parser is used when a query is made against one or more dBASE or Paradox tables, or one or more remote server tables when those table names in the query are preceded by a BDE database alias. The following sections describe the restrictions that must be met to return a live result set for local SQL.

The following topic is discussed in this section:

- Restrictions on live queries

Restrictions on live queries

Topic groups

A live result set for a query against a single table or view is returned if the query does not contain any of the following:

- DISTINCT in the SELECT clause
- Joins (inner, outer, or UNION)
- Aggregate functions with or without GROUP BY or HAVING clauses
- Base tables or views that are not updatable
- Subqueries
- ORDER BY clauses not based on an index

Remote server SQL requirements for a live result set

Topic groups See also

For queries that use passthrough SQL, which includes all queries made solely against remote database servers, live result sets are restricted to the standards defined by SQL-92 and any additional, server-imposed restrictions.

A live result set for a query against a single table or view is returned if the query does not contain any of the following:

- A DISTINCT clause in the SELECT statement
- Aggregate functions, with or without GROUP BY or HAVING clauses
- References to more than one base table or updatable views (joins)
- Subqueries that reference the table in the FROM clause or other tables

Restrictions on updating a live result set

Topic groups See also

If a query returns a live result set, you may not be able to update the result set directly if the result set contains linked fields or you switch indexes before attempting an update. If these conditions exist, you may be able to treat the result set as a read-only result set, and update it accordingly.

Updating a read-only result set

Topic groups See also

Applications can update data returned in a read-only result set if they are using cached updates. To update a read-only result set associated with a query component:

1. Add a *TUpdateSQL* component to the data module in your application to essentially give you the ability to post updates to a read-only dataset.
2. Enter the SQL update statement for the result set to the update component's *ModifySQL*, *InsertSQL*, or *DeleteSQL* properties.
3. Set the query component's *CachedUpdate* property to *True*.

For more information about using cached updates, see Working with cached updates.

Working with stored procedures

Topic groups

This section describes how to use stored procedures in your database applications. A stored procedure is a self-contained program written in the procedure and trigger language specific to the database system used. There are two fundamental types of stored procedures. The first type retrieves data (like with a SELECT query). The retrieved data can be in the form of a dataset consisting of one or more rows of data, divided into one or more columns. Or the retrieved data can be in the form of individual pieces of information. The second type does not return data, but performs an action on data stored in the database (like with a DELETE statement). Some database servers support performing both types of operations in the same procedure.

Stored procedures that return data do so in different ways, depending on how the stored procedure is composed and the database system used. Some, like InterBase, return all data (datasets and individual pieces of information) exclusively with output parameters. Others are capable of returning a cursor to data. And still others, like Microsoft SQL Server and Sybase, can return a dataset and individual pieces of information.

In Delphi applications, access to stored procedures is provided by the *TStoredProc* and *TQuery* components. The choice of which to use for the access is predicated on how the stored procedure is coded, how data is returned (if any), and the database system used. The *TStoredProc* and *TQuery* components are both descendants of *TDataSet*, and inherit behaviors from *TDataSet*. For more information about *TDataSet*, see Understanding datasets.

A stored procedure component is used to execute stored procedures that do not return any data, to retrieve individual pieces of information in the form of output parameters, and to relay a returned dataset to an associated data source component (this last being database-specific). The stored procedure component allows values to be passed to and return from the stored procedure through parameters, each parameter defined in the *Params* property. The stored procedure component also provides a *GetResults* method to force the stored procedure to return a dataset (some database servers require this before a result set is produced). The stored procedure component is the preferred means for using stored procedures that either do not return any data or only return data through output parameters.

A query component is primarily used to run stored procedures that return datasets. This includes InterBase stored procedures that only return datasets via output parameters. The query component can also be used to execute a stored procedure that does not return a dataset or output parameter values.

Use parameters to pass distinct values to or return values from a stored procedure. Input parameter values are used in such places as the WHERE clause of a SELECT statement in a stored procedure. An output parameter allows a stored procedure to pass a single value to the calling application. Some stored procedures return a result parameter. See the documentation for the specific database server you are using for details on the types of parameters that are supported and how they are used in the server's procedure language.

The following topics are discussed in this section:

- When should you use stored procedures?
- Using a stored procedure
- Understanding stored procedure parameters
- Working with Oracle overloaded stored procedures

When should you use stored procedures?

Topic groups See also

If your server defines stored procedures, you should use them if they apply to the needs of your application. A database server developer creates stored procedures to handle frequently-repeated database-related tasks. Often, operations that act upon large numbers of rows in database tables—or that use aggregate or mathematical functions—are candidates for stored procedures. If stored procedures exist on the remote database server your application uses, you should take advantage of them in your application. Chances are you need some of the functionality they provide, and you stand to improve the performance of your database application by:

- Taking advantage of the server's usually greater processing power and speed.
- Reducing the amount of network traffic since the processing takes place on the server where the data resides.

For example, consider an application that needs to compute a single value: the standard deviation of values over a large number of records. To perform this function in your application, all the values used in the computation must be fetched from the server, resulting in increased network traffic. Then your application must perform the computation. Because all you want in your application is the end result—a single value representing the standard deviation—it would be far more efficient for a stored procedure on the server to read the data stored there, perform the calculation, and pass your application the single value it requires.

See your server's database documentation for more information about its support for stored procedures.

Using a stored procedure

Topic groups See also

How a stored procedure is used in a Delphi application depends on how the stored procedure was coded, whether and how it returns data, the specific database server used, or a combination of these factors.

In general terms, to access a stored procedure on a server, an application must:

1. Instantiate a *TStoredProc* component and optionally associate it with a stored procedure on the server. Or instantiate a *TQuery* component and compose the contents of its SQL property to perform either a SELECT query against the stored procedure or an EXECUTE command, depending on whether the stored procedure returns a result set. For more information about creating a *TStoredProc*, see Creating a stored procedure component. For more information about creating a *TQuery* component, see Working with queries.
2. Provide input parameter values to the stored procedure component, if necessary. When a stored procedure component is not associated with stored procedure on a server, you must provide additional input parameter information, such as parameter names and data types. For more information about providing input parameter information, see Setting parameter information at design time.
3. Execute the stored procedure.
4. Process any result and output parameters. As with any other dataset component, you can also examine the result dataset returned from the server. For more information about output and result parameters, see Using output parameters. and Using the result parameter. For information about viewing records in a dataset, see Using stored procedures that return result sets.

Detailed aspects of using stored procedures include:

- Creating a stored procedure component
- Creating a stored procedure
- Preparing and executing a stored procedure
- Using stored procedures that return result sets
- Using stored procedures that return data using parameters
- Using stored procedures that perform actions on data

Creating a stored procedure component

Topic groups See also

To create a stored procedure component for a stored procedure on a database server:

1. Place a stored procedure component from the Data Access page of the Component palette in a data module.
2. Optionally set the *DatabaseName* property of the stored procedure component to the name of the database in which the stored procedure is defined. *DatabaseName* must be a BDE alias or the same value as in the *DatabaseName* property of a *TDatabase* that can connect to the server.

   Normally you should specify the *DatabaseName* property, but if the server database against which your application runs is currently unavailable, you can still create and set up a stored procedure component by omitting the *DatabaseName* and supplying a stored procedure name and input, output, and result parameters at design time. For more information about input parameters, see Using input parameters. For more information about output parameters, see Using output parameters. For more information about the result parameter, see Using the result parameter.
3. Optionally set the *StoredProcName* property to the name of the stored procedure to use. If you provided a value for the *DatabaseName* property, then you can select a stored procedure name from the drop-down list for the property. A single *TStoredProc* component can be used to execute any number of stored procedures by setting the *StoredProcName* property to a valid name in the application. It may not always be desirable to set the *StoredProcName* at design time.
4. Double-click the *Params* property value box to invoke the StoredProc Parameters editor to examine input and output parameters for the stored procedure. If you did not specify a name for the stored procedure in Step 3, or you specified a name for the stored procedure that does not exist on the server specified in the *DatabaseName* property in Step 2, then when you invoke the parameters editor, it is empty.

Not all servers return parameters or parameter information. See your server's documentation to determine what information about its stored procedures it returns to client applications.

Note: If you do not specify the *DatabaseName* property in Step 2, then you must use the StoredProc Parameters editor to set up parameters at design time. For information about setting parameters at design time, see Setting parameter information at design time.

Creating a stored procedure

Topic groups Example

Ordinarily, stored procedures are created when the application and its database is created, using tools supplied by the database system vendor. However, it is possible to create stored procedures at runtime. The specific SQL statement used will vary from one database system to another because the procedure language varies so greatly. Consult the documentation for the specific database system you are using for the procedure language that is supported.

A stored procedure can be created by an application at runtime using an SQL statement issued from a *TQuery* component, typically with a CREATE PROCEDURE statement. If parameters are used in the stored procedure, set the *ParamCheck* property of the *TQuery* to *False*. This prevents the *TQuery* from mistaking the parameter in the new stored procedure from a parameter for the *TQuery* itself.

Note: You can also use the SQL Explorer to examine, edit, and create stored procedures on the server.

Example: Creating a stored procedure with a TQuery

After the SQL property has been populated with the statement to create the stored procedure, execute it by invoking the *ExecSQL* method.

```
with Query1 do begin
  ParamCheck := False;
  with SQL do begin
    Clear;
    Add('CREATE PROCEDURE GET_MAX_EMP_NAME');
    Add('RETURNS (Max_Name CHAR(15))');
    Add('AS');
    Add('BEGIN');
    Add('  SELECT MAX(LAST_NAME)');
    Add('  FROM EMPLOYEE');
    Add('  INTO :Max_Name;');
    Add('  SUSPEND;');
    Add('END');
  end;
  ExecSQL;
end;
```

Preparing and executing a stored procedure

Topic groups See also

To use a stored procedure, you can optionally prepare it, and then execute it.

You can prepare a stored procedure at:

- Design time, by choosing OK in the Parameters editor.
- Runtime, by calling the *Prepare* method of the stored procedure component.

For example, the following code prepares a stored procedure for execution:

```
StoredProc1.Prepare;
```

Note: If your application changes parameter information at runtime, such as when using Oracle overloaded procedures, you should prepare the procedure again.

To execute a prepared stored procedure, call the *ExecProc* method for the stored procedure component. The following code illustrates code that prepares and executes a stored procedure:

```
StoredProc1.Params[0].AsString := Edit1.Text;
StoredProc1.Prepare;
StoredProc1.ExecProc;
```

Note: If you attempt to execute a stored procedure before preparing it, the stored procedure component automatically prepares it for you, and then unprepares it after it executes. If you plan to execute a stored procedure a number of times, it is more efficient to call *Prepare* yourself, and then only call *UnPrepare* once, when you no longer need to execute the procedure.

When you execute a stored procedure, it can return all or some of these items:

- A dataset consisting of one or more records that can be viewed in data-aware controls associated with the stored procedure through a data source component.
- Output parameters.
- A result parameter that contains status information about the stored procedure's execution.

To determine the return items to expect from a stored procedure on your server, see your server's documentation.

Stored procedures that return result sets

Topic groups

Stored procedures that return data in datasets, rows and columns of data, should most often be used with a query component. However, with database servers that support returning a dataset by a stored procedure, a stored procedure component can serve this purpose.

- Retrieving a result set with a TQuery
- Retrieving a result set with a TStoredProc

Retrieving a result set with a TQuery

Topic groups Example

To retrieve a dataset from a stored procedure using a *TQuery* component:

1. Instantiate a query component.
2. In the *TQuery.SQL* property, write a SELECT query that uses the name of the stored procedure instead of a table name.
3. If the stored procedure requires input parameters, express the parameter values as a comma-separated list, enclosed in parentheses, following the procedure name.
4. Set the *Active* property to *True* or invoke the *Open* method.

Example: Retrieving a dataset from a stored procedure with a TQuery

For example, the InterBase stored procedure GET_EMP_PROJ, below, accepts a value using the input parameter EMP_NO and returns a dataset through the output parameter PROJ_ID.

```
CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID CHAR(5))
AS
BEGIN
  FOR SELECT PROJ_ID
  FROM EMPLOYEE_PROJECT
  WHERE EMP_NO = :EMP_NO
  INTO :PROJ_ID
  DO
    SUSPEND;
END
```

The SQL statement issued from a *TQuery* to use this stored procedure would be:

```
SELECT *
FROM GET_EMP_PROJ(52)
```

Retrieving a result set with a TStoredProc

Topic groups Example

To retrieve a dataset from a stored procedure using a *TStoredProc* component:

1. Instantiate a stored procedure component.
2. In the *StoredProcName* property, specify the name of the stored procedure.
3. If the stored procedure requires input parameters, supply values for the parameters using the *Params* property or *ParamByName* method.
4. Set the *Active* property to *True* or invoke the *Open* method.

Example: Retrieving a dataset from a stored procedure with a TStoredProc

For example, the Sybase stored procedure GET_EMPLOYEES, below, accepts an input parameter named @EMP_NO and returns a result set based on that value.

```
CREATE PROCEDURE GET_EMPLOYEES @EMP_NO SMALLINT
AS SELECT EMP_NAME, EMPLOYEE_NO FROM EMPLOYEE_TABLE
WHERE (EMPLOYEE_NO = @EMP_NO)
```

The Delphi code to fill the parameter with a value and activate the stored procedure component is:

```
with StoredProc1 do begin
  Close;
  ParamByName('EMP_NO').AsSmallInt := 52;
  Active := True;
end;
```

Stored procedures that return data using parameters

Topic groups

Stored procedures can be composed to retrieve individual pieces of information, as opposed to whole rows of data, through parameters. For instance, a stored procedure might retrieve the maximum value for a column, add one to that value, and then return that value to the application. Such stored procedures can be used and the values inspected using either a *TQuery* or a *TStoredProc* component. The preferred method for retrieving parameter values is with a *TStoredProc*.

- Retrieving individual values with a TQuery
- Retrieving individual values with a TStoredProc

Retrieving individual values with a TQuery

Topic groups Example

Parameter values retrieved via a *TQuery* component take the form of a single-row dataset, even if only one parameter is returned by the stored procedure. To retrieve individual values from stored procedure parameters using a *TQuery* component:

1. Instantiate a query component.
2. In the *TQuery.SQL* property, write a SELECT query that uses the name of the stored procedure instead of a table name. The SELECT clause of this query can specify the parameter by its name, as if it were a column in a table, or it can simply use the * operator to retrieve all parameter values.
3. If the stored procedure requires input parameters, express the parameter values as a comma-separated list, enclosed in parentheses, following the procedure name.
4. Set the *Active* property to *True* or invoke the *Open* method.

Example: Retrieving stored procedure parameter values with a TQuery

For example, the InterBase stored procedure GET_HIGH_EMP_NAME, below, retrieves the alphabetically last value in the LAST_NAME column of a table named EMPLOYEE. The stored procedure returns this value in the output parameter High_Last_Name.

```
CREATE PROCEDURE GET_HIGH_EMP_NAME
RETURNS (High_Last_Name CHAR(15))
AS
BEGIN
  SELECT MAX(LAST_NAME)
  FROM EMPLOYEE
  INTO :High_Last_Name;
  SUSPEND;
END
```

The SQL statement issued from a *TQuery* to use this stored procedure would be:

```
SELECT High_Last_Name
FROM GET_HIGH_EMP_NAME
```

Retrieving individual values with a TStoredProc

Topic groups Example

To retrieve individual values from stored procedure output parameters using a *TStoredProc* component:

1. Instantiate a stored procedure component.
2. In the *StoredProcName* property, specify the name of the stored procedure.
3. If the stored procedure requires input parameters, supply values for the parameters using the *Params* property or *ParamByName* method.
4. Invoke the *ExecProc* method.
5. Inspect the values of individual output parameters using the *Params* property or *ParamByName* method.

Example: Retrieving output parameter values with a TStoredProc

For example, the InterBase stored procedure GET_HIGH_EMP_NAME, below, retrieves the alphabetically last value in the LAST_NAME column of a table named EMPLOYEE. The stored procedure returns this value in the output parameter High_Last_Name.

```
CREATE PROCEDURE GET_HIGH_EMP_NAME
RETURNS (High_Last_Name CHAR(15))
AS
BEGIN
  SELECT MAX(LAST_NAME)
  FROM EMPLOYEE
  INTO :High_Last_Name;
  SUSPEND;
END
```

The Delphi code to get the value in the High_Last_Name output parameter and store it to the *Text* property of a *TEdit* component is:

```
with StoredProc1 do begin
  StoredProcName := 'GET_HIGH_EMP_NAME';
  ExecProc;
  Edit1.Text := ParamByName('High_Last_Name').AsString;
end;
```

Stored procedures that perform actions on data

Topic groups

Stored procedures can be coded such that they do not return any data at all, and only perform some action in the database. SQL operations involving the INSERT and DELETE statements are good examples of this type of stored procedure. For instance, instead of allowing a user to delete a row directly, a stored procedure might be used to do so. This would allow the stored procedure to control what is deleted and also to handle any referential integrity aspects, such as a cascading delete of rows in dependent tables.

- Executing an action stored procedure with a TQuery
- Executing an action stored procedure with a TStoredProc

Executing an action stored procedure with a TQuery

Topic groups Example

To execute an action stored procedure using a *TQuery* component:

1. Instantiate a query component.
2. In the *TQuery.SQL* property, include the command necessary to execute the stored procedure and the stored procedure name. (The command to execute a stored procedure can vary from one database system to another. In InterBase, the command is EXECUTE PROCEDURE.)
3. If the stored procedure requires input parameters, express the parameter values as a comma-separated list, enclosed in parentheses, following the procedure name.
4. Invoke the *TQuery.ExecSQL* method.

Example: Executing an action stored procedure with a TQuery

For example, the InterBase stored procedure ADD_EMP_PROJ, below, adds a new row to the table EMPLOYEE_PROJECT. No dataset is returned and no individual values are returned in output parameters.

```
CREATE PROCEDURE ADD_EMP_PROJ (EMP_NO SMALLINT, PROJ_ID CHAR(5))
AS
BEGIN
  BEGIN
    INSERT INTO EMPLOYEE_PROJECT (EMP_NO, PROJ_ID)
    VALUES (:EMP_NO, :PROJ_ID);
    WHEN SQLCODE -530 DO
      EXCEPTION UNKNOWN_EMP_ID;
  END
  SUSPEND;
END
```

The SQL statement issued from a *TQuery* to execute this stored procedure would be:

```
EXECUTE PROCEDURE ADD_EMP_PROJ(20, "GUIDE")
```

Executing an action stored procedure with a TStoredProc

Topic groups See also Example

To retrieve individual values from stored procedure output parameters using a *TStoredProc* component:

1. Instantiate a stored procedure component.
2. In the *StoredProcName* property, specify the name of the stored procedure.
3. If the stored procedure requires input parameters, supply values for the parameters using the *Params* property or *ParamByName* method.
4. Invoke the *ExecProc* method.

Example: Executing an action stored procedure with a TStoredProc

For example, the InterBase stored procedure ADD_EMP_PROJ, below, adds a new row to the table EMPLOYEE_PROJECT. No dataset is returned and no individual values are returned in output parameters.

```
CREATE PROCEDURE ADD_EMP_PROJ (EMP_NO SMALLINT, PROJ_ID CHAR(5))
AS
BEGIN
  BEGIN
    INSERT INTO EMPLOYEE_PROJECT (EMP_NO, PROJ_ID)
    VALUES (:EMP_NO, :PROJ_ID);
    WHEN SQLCODE -530 DO
      EXCEPTION UNKNOWN_EMP_ID;
  END
  SUSPEND;
END
```

The Delphi code to execute the ADD_EMP_PROJ stored procedure is:

```
with StoredProc1 do begin
  StoredProcName := 'ADD_EMP_PROJ';
  ExecProc;
end;
```

Understanding stored procedure parameters

Topic groups See also

There are four types of parameters that can be associated with stored procedures:

- *Input parameters*, used to pass values to a stored procedure for processing.
- *Output parameters*, used by a stored procedure to pass return values to an application.
- *Input/output parameters*, used to pass values to a stored procedure for processing, and used by the stored procedure to pass return values to the application.
- A *result parameter*, used by some stored procedures to return an error or status value to an application. A stored procedure can only return one result parameter.

Whether a stored procedure uses a particular type of parameter depends both on the general language implementation of stored procedures on your database server and on a specific instance of a stored procedure. For example, individual stored procedures on any server may either be implemented using input parameters, or may not be. On the other hand, some uses of parameters are server-specific. For example, on MS-SQL Server and Sybase stored procedures always return a result parameter, but the InterBase implementation of a stored procedure never returns a result parameter.

Access to stored procedure parameters is provided by *TParam* objects in the *TStoredProc.Params* property. If the name of the stored procedure is specified at design time in the *StoredProcName* property, a *TParam* object is automatically created for each parameter and added to the *Params* property. If the stored procedure name is not specified until runtime, the *TParam* objects need to be programmatically created at that time. Not specifying the stored procedure and manually creating the *TParam* objects allows a single *TStoredProc* component to be used with any number of available stored procedures.

Note: Some stored procedures return a dataset in addition to output and result parameters. Applications can display dataset records in data-aware controls, but must separately process output and result parameters. For more information about displaying records in data-aware controls, see Using stored procedures that return result sets

The following topics are discussed in this section:

- Using input parameters
- Using output parameters
- Using input/output parameters
- Using the result parameter
- Accessing parameters at design time
- Creating parameters at runtime

Using input parameters

Topic groups See also

Application use input parameters to pass singleton data values to a stored procedure. Such values are then used in SQL statements within the stored procedure, such as a comparison value for a WHERE clause. If a stored procedure requires an input parameter, assign a value to the parameter prior to executing the stored procedure.

If a stored procedure returns a dataset and is used through a SELECT query in a *TQuery* component, supply input parameter values as a comma-separated list, enclosed in parentheses, following the stored procedure name. For example, the SQL statement below retrieves data from a stored procedure named GET_EMP_PROJ and supplies an input parameter value of 52.

```
SELECT PROJ_ID
FROM GET_EMP_PROJ(52)
```

If a stored procedure is executed with a *TStoredProc* component, use the *Params* property or the *ParamByName* method access to set each input parameter. Use the *TParam* property appropriate for the data type of the parameter, such as the *TParam.AsString* property for a CHAR type parameter. Set input parameter values prior to executing or activating the *TStoredProc* component. In the example below, the EMP_NO parameter (type SMALLINT) for the stored procedure GET_EMP_PROJ is assigned the value 52.

```
with StoredProc1 do begin
  ParamByName('EMP_NO').AsSmallInt := 52;
  ExecProc;
end;
```

Using output parameters

Topic groups See also

Stored procedures use output parameters to pass singleton data values to an application that calls the stored procedure. Output parameters are not assigned values except by the stored procedure and then only after the stored procedure has been executed. Inspect output parameters from an application to retrieve its value after invoking the *TStoredProc.ExecProc* method.

Use the *TStoredProc.Params* property or *TStoredProc.ParamByName* method to reference the *TParam* object that represents a parameter and inspect its value. For example, to retrieve the value of a parameter and store it into the *Text* property of a *TEdit* component:

```
with StoredProc1 do begin
  ExecProc;
  Edit1.Text := Params[0].AsString;
end;
```

Most stored procedures return one or more output parameters. Output parameters may represent the sole return values for a stored procedure that does not also return a dataset, they may represent one set of values returned by a procedure that also returns a dataset, or they may represent values that have no direct correspondence to an individual record in the dataset returned by the stored procedure. Each server's implementation of stored procedures differs in this regard.

Note: The source code for an Informix stored procedure may indicate that it returns output parameters even though you cannot not see output parameter information in the StoredProc Parameters editor. Informix translates output parameters into a single record dataset that you can view in your application's data-aware controls.

Using input/output parameters

Topic groups See also

Input/output parameters serve both function that input and output parameters serve individually. Applications use an input/output parameter to pass a singleton data value to a stored procedure, which in turn reuses the input/output parameter to pass a singleton data value to the calling application. As with input parameters, the input value for an input/output parameter must be set before the using stored procedure or query component is activated. Likewise, the output value in an input/output parameter will not be available until after the stored procedure has been executed.

In the example Oracle stored procedure below, the parameter IN_OUTVAR is an input/output parameter.

```
CREATE OR REPLACE PROCEDURE UPDATE_THE_TABLE (IN_OUTVAR IN OUT INTEGER)
AS
BEGIN
  UPDATE ALLTYPETABLE
  SET NUMBER82FLD = IN_OUTVAR
  WHERE KEYFIELD = 0;
  IN_OUTVAR:=1;
END UPDATE_THE_TABLE;
```

In the Delphi program code below, IN_OUTVAR is assigned an input value, the stored procedure executed, and then the output value in IN_OUTVAR is inspected and stored to a memory variable.

```
with StoredProc1 do begin
  ParamByName('IN_OUTVAR').AsInteger := 103;
  ExecProc;
  IntegerVar := ParamByName('IN_OUTVAR').AsInteger;
end;
```

Using the result parameter

Topic groups See also

In addition to returning output parameters and a dataset, some stored procedures also return a single result parameter. The result parameter is usually used to indicate an error status or the number of records processed base on stored procedure execution. See your database server's documentation to determine if and how your server supports the result parameter. Result parameters are not assigned values except by the stored procedure and then only after the stored procedure has been executed. Inspect a result parameter from an application to retrieve its value after invoking the *TStoredProc.ExecProc* method.

Use the *TStoredProc.Params* property or *TStoredProc.ParamByName* method to reference the *TParam* object that represents the result parameter and inspect its value.

```
DateVar := StoredProc1.ParamByName('MyOutputParam').AsDate;
```

Accessing parameters at design time

Topic groups See also

If you connect to a remote database server by setting the *DatabaseName* and *StoredProcName* properties at design time, then you can use the StoredProc Parameters editor to view the names and data types of each input parameter, and you can set the values for the input parameters to pass to the server when you execute the stored procedure.

Important: Do not change the names or data types for input parameters reported by the server, or when you execute the stored procedure an exception is raised.

Some servers—Informix, for example—do not report parameter names or data types. In these cases, use the SQL Explorer or vendor-supplied server utilities to look at the source code of the stored procedure on the server to determine input parameters and data types. See the SQL Explorer online help for more information.

At design time, if you do not receive a parameter list from a stored procedure on a remote server (for example because you are not connected to a server), then you must invoke the StoredProc Parameters editor, list each required input parameter, and assign each a data type and a value. For more information about using the StoredProc Parameters editor to create parameters, see Setting parameter information at design time

Setting parameter information at design time

Topic groups See also

You can invoke the StoredProc parameter collection editor at design time to set up parameters and their values.

The parameter collection editor allows you to set up stored procedure parameters. If you set the *DatabaseName* and *StoredProcName* properties of the *TStoredProc* component at design time, all existing parameters are listed in the collection editor. If you do not set both of these properties, no parameters are listed and you must add them manually. Additionally, some database types do not return all parameter information, like types. For these database systems, use the SQL Explorer utility to inspect the stored procedures, determine types, and then configure parameters through the collection editor and the Object Inspector. The steps to set up stored procedure parameters at design time are:

1. Optionally set the *DatabaseName* and *StoredProcName* properties.
2. In the Object Inspector, invoke the parameter collection editor by clicking on the ellipsis button in the *Params* field.
3. If the *DatabaseName* and *StoredProcName* properties are not set, no parameters appear in the collection editor. Manually add parameter definitions by right-clicking within the collection editor and selecting Add from the context menu.
4. Select parameters individually in the collection editor to display their properties in the Object Inspector.
5. If a type is not automatically specified for the *ParamType* property, select a parameter type (*Input*, *Output*, *Input/Output*, or *Result*) from the property's drop-down list.
6. If a data type is not automatically specified for the *DataType* property, select a data type from the property's drop-down list. (To return a result set from an Oracle stored procedure, set field type to Cursor.)
7. Use the *Value* property to optionally specify a starting value for an input or input/output parameter.

Right-clicking in the parameter collection editor invokes a context menu for operating on parameter definitions. Depending on whether any parameters are listed or selected, enabled options include: adding new parameters, deleting existing parameters, moving parameters up and down in the list, and selecting all listed parameters.

You can edit the definition for any *TParam* you add, but the attributes of the *TParam* objects you add must match the attributes of the parameters for the stored procedure on the server. To edit the *TParam* for a parameter, select it in the parameter collection editor and edit its property values in the Object Inspector.

Note: Sybase, MS-SQL, and Informix do not return parameter type information. Use the SQL Explorer to determine this information.

Note: Informix does not return data type information. Use the SQL Explorer to determine this information.

Note: You can never set values for output and result parameters. These types of parameters have values set by the execution of the stored procedure.

Creating parameters at runtime

Topic groups See also

If the name of the stored procedure is not specified in StoredProcName until runtime, no *TParam* objects will be automatically created for parameters and they must be created programmatically. This can be done using the *TParam.Create* method or the *TParams.AddParam* method.

For example, the InterBase stored procedure GET_EMP_PROJ, below, requires one input parameter (EMP_NO) and one output parameter (PROJ_ID).

```
CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID CHAR(5))
AS
BEGIN
  FOR SELECT PROJ_ID
  FROM EMPLOYEE_PROJECT
  WHERE EMP_NO = :EMP_NO
  INTO :PROJ_ID
  DO
    SUSPEND;
END
```

The Delphi code to associate this stored procedure with a *TStoredProc* named *StoredProc1* and create *TParam* objects for the two parameters using the *TParam.Create* method is:

```
var
  P1, P2: TParam;
begin
  ...
  with StoredProc1 do begin
    StoredProcName := 'GET_EMP_PROJ';
    Params.Clear;
    P1 := TParam.Create(Params, ptInput);
    P2 := TParam.Create(Params, ptOutput);
    try
      Params[0].Name := 'EMP_NO';
      Params[1].Name := 'PROJ_ID';
      ParamByname('EMP_NO').AsSmallInt := 52;
      ExecProc;
      Edit1.Text := ParamByname('PROJ_ID').AsString;
    finally
      P1.Free;
      P2.Free;
    end;
  end;
  ...
end;
```

Binding parameters

Topic groups See also

When you prepare and execute a stored procedure, its input parameters are automatically bound to parameters on the server.

Use the *ParamBindMode* property to specify how parameters in your stored procedure component should be bound to the parameters on the server. By default *ParamBindMode* is set to *pbByName*, meaning that parameters from the stored procedure component are matched to those on the server by name. This is the easiest method of binding parameters.

Some servers also support binding parameters by ordinal value, the order in which the parameters appear in the stored procedure. In this case the order in which you specify parameters in the parameter collection editor is significant. The first parameter you specify is matched to the first input parameter on the server, the second parameter is matched to the second input parameter on the server, and so on. If your server supports parameter binding by ordinal value, you can set *ParamBindMode* to *pbByNumber*.

Tip: If you want to set *ParamBindMode* to *pbByNumber*, you need to specify the correct parameter types in the correct order. You can view a server's stored procedure source code in the SQL Explorer to determine the correct order and type of parameters to specify.

Viewing parameter information at design time

Topic groups See also

If you have access to a database server at design time, there are two ways to view information about the parameters used by a stored procedure:

- Invoke the SQL Explorer to view the source code for a stored procedure on a remote server. The source code includes parameter declarations that identify the data types and names for each parameter.
- Use the Object Inspector to view the property settings for individual *TParam* objects.

You can use the SQL Explorer to examine stored procedures on your database servers if you are using BDE native drivers. If you are using ODBC drivers you cannot examine stored procedures with the SQL Explorer. While using the SQL Explorer is not always an option, it can sometimes provide more information than the Object Inspector viewing *TParam* objects. The amount of information returned about a stored procedure in the Object Inspector depends on your database server.

To view individual parameter definitions in the Object Inspector:

1. Select the stored procedure component.
2. Set the *DatabaseName* property of a stored procedure component to the BDE alias for your database server (or the *DatabaseName* property of a *TDatabase*).
3. Set the *StoredProcName* property to the name of the stored procedure.
4. click the ellipsis button in for the *TStoredProc.Params* property in the Object Inspector.
5. Select individual parameters in the collection editor to view their property settings in the Object Inspector.

For some servers some or all parameter information may not be accessible.

In the Object Inspector, when viewing individual *TParam* objects, the *ParamType* property indicates whether the selected parameter is an input, output, input/output, or result parameter. The *DataType* property indicates the data type of the value the parameter contains, such as string, integer, or date. The *Value* edit box enables you to enter a value for a selected input parameter.

Note: Sybase, MS-SQL, and Informix do not return parameter type information. Use the SQL Explorer or vendor-supplied server utilities to determine this information.

Note: Informix does not return data type information. Use the SQL Explorer vendor-supplied server utilities to determine this information.

For more about setting parameter values, see Setting parameter information at design time

Note: You can never set values for output and result parameters. These types of parameters have values set by the execution of the stored procedure.

Working with Oracle overloaded stored procedures

Topic groups

Oracle servers allow overloading of stored procedures; overloaded procedures are different procedures with the same name. The stored procedure component's *Overload* property enables an application to specify the procedure to execute.

If *Overload* is zero (the default), there is assumed to be no overloading. If *Overload* is one (1), then the stored procedure component executes the first stored procedure it finds on the Oracle server that has the overloaded name; if it is two (2), it executes the second, and so on.

Note: Overloaded stored procedures may take different input and output parameters. See your Oracle server documentation for more information.

Working with ADO components

Topic groups See also

The ADO components encapsulate the functionality of the ADO framework. ADO, or Microsoft ActiveX Data Objects, is a set of data objects that provide an application the ability to access data through an OLE DB provider. The Delphi ADO components encapsulate the functionality of these ADO objects and present their functionality in the context of Delphi components. The ADO objects that figure most prominently are the Connection, Command, and Recordset objects. These ADO objects are directly represented in the *TADOConnection*, *TADOCommand*, and ADO dataset components. There are other "helper" objects in the ADO framework, like the Field and Properties objects, but they are generally not used directly by the Delphi programmer and not represented by dedicated components.

Using ADO and the ADO components allows the Delphi programmer to create database applications that are not dependent on the Borland Database Engine (BDE), using instead ADO for the data access.

This section presents each of the ADO components and discusses how they differ from their BDE-based counterparts. References are given to topics covering aspects of the BDE-based connection and dataset components that are the same in the ADO equivalents.

The general areas covered in this chapter include:

- Overview of ADO components
- Connecting to ADO data stores
- Using ADO datasets
- Executing commands

Overview of ADO components

Topic groups See also

In addition to the connection and dataset components based on the Borland Database Engine (BDE), Delphi provides a set of components for use with ADO. These components allow the programmer to connect to an ADO data store and then to execute commands and retrieve data from tables in databases.

These ADO-centric data access components connect to ADO data stores and operate on data using only the ADO framework. The BDE is not employed at all in this process. Use the ADO components when ADO is available and you do not want to use the BDE. ADO 2.1 (or higher) must be installed on the host computer. Additionally, client software for the target database system (such as Microsoft SQL Server) must be installed as well as an OLE DB driver or ODBC driver specific to the particular database system.

Most of the ADO connection and dataset components are analogous to one of the BDE-based connection or dataset components. The *TADOConnection* component is functionally analogous to the *TDatabase* component in BDE-based applications. *TADOTable* is equivalent to *TTable*, *TADOQuery* to *TQuery*, and *TADOStoredProc* to *TStoredProc*. Use these ADO components in the same manner and context as you would the BDE-based data equivalents. *TADODataSet* has no direct BDE equivalent, but provides many of the same functions as *TTable* and *TQuery*. Similarly, there is no BDE component comparable to *TADOCommand*, which serves a specialized purpose in the Delphi/ADO environment.

The ADO components comprise the following classes.

Component	Use
TADOConnection	Used to establish a connection with an ADO data store; multiple ADO dataset and command components can share this connection to execute commands, retrieve data, and to operate on metadata.
TADODataSet	The primary component used to retrieve and operate on its data; can retrieve data from a single or multiple tables; can connect directly to a data store or through a *TADOConnection*.
TADOTable	Used to retrieve and operate on a dataset produced by a single table; can connect directly to a data store or through a *TADOConnection*.
TADOQuery	Used to retrieve and operate on a dataset produced by a valid SQL statement; can also execute data definition language (DDL) SQL statements, like CREATE TABLE; can connect directly to a data store or through a *TADOConnection*.
TADOStoredProc	Used to execute stored procedures; can execute stored procedures that retrieve data or execute DDL statements; can connect directly to a data store or through a *TADOConnection*.
TADOCommand	Used primarily to execute commands (SQL statements that do not return result sets); used with a supporting dataset component, can also retrieve a dataset from a table; can connect directly to a data store or through a *TADOConnection*.

Connecting to ADO data stores

Topic groups See also

Before commands can be executed or data retrieved, an application must establish a connection to a data store. While each individual ADO command and dataset component in an application can establish its own connection, a *TADOConnection* can be used and its single connection shared by other ADO components.

When connecting ADO command and dataset components to a data store, they can all use a shared connection or the components can each establish their own connections. Each approach has its own advantages and disadvantages.

This section covers the tasks involved in establishing and using a connection to an ADO data store. This includes the major subject areas:

- Connecting to a data store using TADOConnection
- Fine-tuning a connection
- Listing tables and stored procedures
- Working with (connection) transactions

Connecting to a data store

Topic groups See also

One or more ADO dataset and command components can share a single connection to a data store. To do this, the application must have one *TADOConnection* component to make each data store connection. Then, the dataset and command components are associated with the connection component through their *Connection* properties.

In addition to providing the means for dataset and command components to connect to a data store, connection components provide properties and methods for activating and deactivating the connection, accessing the ADO connection object directly, and for determining what activity (if any) a connection component is engaged in at any given time.

- Using a TADOConnection versus a dataset's ConnectionString
- Specifying the connection
- Accessing the connection object
- Activating and deactivating the connection
- Determining what a connection component is doing

Using a TADOConnection versus a dataset's ConnectionString

Topic groups See also

Each ADO command and dataset component in an application may be connected directly to a data store. However, when numerous command and dataset components are used, it is most often easier to maintain the connection using a single *TADOConnection* to establish the connection and then sharing that connection between the command and dataset components. See the section Connecting to a data store using ADO dataset components for more information on connecting individual command and dataset directly to a data store.

Using a *TADOConnection* component to establish the connection offers more control over the connection versus connecting each command or dataset component individually. This greater control is provided by the properties, methods, and event of the *TADOConnection*, the functionality of which is not available otherwise.

Specifying the connection

Topic groups See also

To use a *TADOConnection* component to supply a shared connection for ADO dataset and command components, first establish the connection. Do this by supplying specific connection information in the *ConnectionString* property of the connection component. At design-time, invoke the connection string editor dialog by clicking the ellipsis button for the *ConnectionString* property in the Object Inspector. This dialog, supplied by the ADO system itself, allows you to interactively build a connection string by selecting connection elements (like the provider and server) from lists. At runtime, assign a **String** value to the *ConnectionString* property with the connection information. Setting the *Connected* property of the connection component to True would activate the connection. However, it is not essential to do so at this point. At design-time this is a good test of the connection, though.

```
ADOConnection1.ConnectionString := 'Provider=ProviderName;Remote
Server=ServerReference';
```

The *ConnectionString* property can contain a number of connection parameters, each separated by semi-colons. These parameters can include the name of a provider, a user name and password (for login purposes), and a reference identifying a remote server. The *ConnectionString* property can also contain the name of a file containing the connection parameters. Such a file has the same contents as the *ConnectionString* property: one or more parameters, each with a value assignment and separated from other parameters by a semi-colon. See the VCL help topic for the ConnectionString property for a list of ADO-supported parameters.

Once the connection information has been provided in the connection component, associate dataset and command components with the connection component. Do this by assigning a reference to the connection component to each dataset or command component's *Connection* property. At design-time, select the desired connection component from the drop-down list for the *Connection* property in the Object Inspector. At runtime, assign the reference to the Connection property. For example, the command below associates a *TADODataSet* component with a *TADOConnection* component.

```
ADODataSet1.Connection := ADOConnection1;
```

If you do not explicitly activate the connection by setting the connection component's *Connected* property to True, it will happen automatically when the first dataset component is activated or the first time a command is executed with a command component.

Accessing the connection object

Topic groups See also

Use the *ConnectionObject* property of *TADOConnection* to access the underlying ADO connection object directly. Using this reference it is possible to access properties and call methods of the underlying ADO Connection object from an application.

Use of *ConnectionObject* to directly access the underlying ADO Connection object requires a good working knowledge of ADO objects in general and the ADO Connection object in specific. It is not recommended that you use the Connection object directly unless familiar with Connection object operations. Consult the Microsoft Data Access SDK help for specific information on using ADO Connection objects.

Activating and deactivating the connection

Topic groups See also

To activate an ADO connection component, set the *TADOConnection*.*Connected* property to True or call the *TADOConnection*.*Open* method.

```
ADOConnection1.Connected := True;
```

For the connection to be successful, the connection information provided in the *TADOConnection*.*ConnectionString* property must define a valid connection. For more information on providing connection information, see Specifying the connection.

Activating an ADO connection component will trigger the *OnWillConnect* and *OnConnectComplete* events of the ADO connection component and execute handlers for these events if they have been assigned.

If a connection component has not already been activated, it will automatically be activated if an associated dataset or command component is enabled. Dataset components cause this when they are activated. Command components do this when a command is executed. For information on associating dataset components with a connection component, see Connecting to a data store using TADOConnection.

To deactivate an ADO connection component, either set its *Active* property to False or call its *Close* method.

```
ADOConnection1.Close;
```

Four things happen when a connection component is deactivated, using either the *Active* property or the *Close* method:

1. The *TADOConnection*.*OnDisconnect* event fires.
2. The handler for the *OnDisconnect* event executes (if one is assigned).
3. The *TADOConnection* component is deactivated.
4. Any associated ADO command or dataset components are deactivated.

Determining what a connection component is doing

Topic groups See also

At any time during the existence of a *TADOConnection* component, query its *State* property to determine what action, if any, in which the connection component is currently engaged.

A *TObjectStates* value of *stClosed* in the *TADOConnection*.*State* property indicates that the connection object is currently inactive. The *TADOConnection*.*Connected* property contains a value of False and no associated command or dataset components are active.

A value of *stOpen* indicates that the connection component is active. A connection with an ADO data store has been successfully established, its *Connected* property contains a value of True, and any one or more associated command or dataset components might be active.

A value of *stConnecting* indicates the connection component is currently attempting to establish a connection to the ADO data store specified in the *TADOConnection*.*ConnectionString* property. While still in this state, the *Cancel* method may be called to abort the connection attempt.

Fine-tuning a connection

Topic groups See also

When a *TADOConnection* component is used to make the connection to a data store for an application's ADO command and dataset components, you have a greater degree of control over the conditions and attributes of the connection. These aspects are implemented using properties and event handlers of *TADOConnection* to fine-tune the connection. These include:

- Specifying connection attributes
- Controlling timeouts
- Controlling the connection login

Specifying connection attributes

Topic groups See also

Use the *TADOConnection.ConnectOptions* property to optionally force the connection to be asynchronous. By default, *ConnectionOptions* is set to *coConnectUnspecified* which allows the server to decide the best type of connection. To explicitly make the connection asynchronous, set *ConnectOptions* to *coAsyncConnect*.

To set up a connection as asynchronous or to delegate the choice to the server, assign one of the *TConnectOption* constants to the connection component's *ConnectOptions* property. Then activate the connection component by calling its *Open* method, setting the *Connected* property to True, or by activating an associated command or dataset component. The example routines below respectively enable and disable asynchronous connections in the specified connection component.

```
procedure TForm1.AsyncConnectButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    ConnectOptions := coAsyncConnect;
    Open;
  end;
end;
procedure TForm1.ServerChoiceConnectButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    ConnectOptions := coConnectUnspecified;
    Open;
  end;
end;
```

Use the *TADOConnection.Attributes* property to control the connection component's use of retaining commits and retaining aborts. *Attributes* can contain one, both, or neither of the constants *xaCommitRetaining* and *xaAbortRetaining*. This makes controlling retaining commits and retaining aborts mutually exclusive using the same property.

Check whether either retaining commits or retaining aborts is enabled using the *in* operator with one of the constants. Enable one feature by adding the constant to the attributes property; disable one by subtracting the constant. The example routines below respectively enable and disable retaining commits in the specified connection component.

```
procedure TForm1.RetainingCommitsOnButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    if not (xaCommitRetaining in Attributes) then
      Attributes := (Attributes + [xaCommitRetaining])
    Open;
  end;
end;
procedure TForm1.RetainingCommitsOffButtonClick(Sender: TObject);
begin
  with ADOConnection1 do begin
    Close;
    if (xaCommitRetaining in Attributes) then
      Attributes := (Attributes - [xaCommitRetaining]);
    Open;
  end;
end;
```

Controlling timeouts

Topic groups See also

Control the period of time before attempted commands and connections are considered failed and are aborted using the *TADOConnection.ConnectionTimeout* property and the *TADOConnection.CommandTimeout* property.

ConnectionTimeout establishes the amount of time before an attempt to connect to the data store times-out. If the connection initiated by a call to the *Open* method has not successfully completed prior to expiration of the time specified in *ConnectionTimeout*, the connection attempt is canceled. Set *ConnectionTimeout* to the number of seconds after which connection attempts time-out.

```
with ADOConnection1 do begin
  ConnectionTimeout = 10 {seconds};
  Open;
end;
```

CommandTimeout establishes the amount of time before attempted commands time-out. If the command initiated by a call to the *Execute* method has not successfully completed prior to expiration of the time specified in *CommandTimeout*, the command is canceled and ADO generates an exception. Set *CommandTimeout* to the number of seconds after which commands time-out.

```
with ADOConnection1 do begin
  CommandTimeout = 10 {seconds};
  Execute('DROP TABLE Employee1997', []);
end;
```

Controlling the connection login

Topic groups See also

An attempt to connect to a data store using a connection component triggers a security login event, *OnLogin*. One manifestation of this event is the appearance of a login dialog prompting for a user name and password. If desired, this dialog may be suppressed and the user name and password information supplied programmatically.

To suppress the default login dialog, first set the *LoginPrompt* property of the connection component to False. Then, prior to activating the connection component, supply all necessary login information via a vehicle such as the *ConnectionString* property.

```
with ADOConnection1 do begin
  Close;
  LoginPrompt := False;
  ConnectionString := 'Provider=NameOfYourProvider;Remote Server=NameOfYourServer;'
+
    'User Name=JaneDoe;Password=SecretWord';
  Connected := True;
end;
```

The login information can also be conveyed to the target data store as parameters for the connection component's *Open* method.

```
with ADOConnection1 do begin
  Close;
  LoginPrompt := False;
  ConnectionString := 'Provider=NameOfYourProvider;Remote Server=NameOfYourServer';
  Open('JaneDoe', 'SecretWord');
end;
```

The second routine above is functionally equivalent to the first. The difference is that in the second routine the user name and password are not expressed in the *ConnectionString* property, but passed as parameters for the *Open* method. This is useful in situations where the connection specifications (like provider and server) are the same for all users and only the information particular to individual users changes. The application might, for instance, obtain the user-specific information via a custom login dialog, and the provider and server information form a static source like Windows Registry entries.

If, after supplying the login information programmatically, the login attempt is unsuccessful, an exception of type *EOleException* is raised.

Working with a connection’s dataset components

Topic groups See also

The *TADOConnection* component provides properties for retrieving lists of the tables and stored procedures available through the connection. It also provides properties for accessing the dataset and command components associated with the connection component. Use of these properties and methods are covered in the sections:

- Accessing the connection’s datasets
- Accessing the connection’s commands
- Listing available tables
- Listing available stored procedures

Accessing the connection's datasets

Topic groups See also

The *DataSets* and *DataSetCount* properties of *TADOConnection* allow a program to sequentially reference each dataset component associated with a connection component. Dataset components operated on by the *DataSets* and *DataSetCount* properties include *TADODataSet*, *TADOQuery*, and *TADOStoredProc*. For working with a connection's command components, use the *Commands* and *CommandCount* properties.

DataSets is a zero-based array of references to ADO dataset components. Use an index with *DataSets* representing the position within the array of a particular dataset. For instance, use an index of 3 to reference the fourth dataset component in *DataSets*.

```
ShowMessage(ADOConnection1.DataSets[3].Name);
```

As *DataSets* provides a reference of type *TCustomADODataSet*, typecast this reference as a descendant class type to access a property or call a method only available in a descendant class. For instance, *TCustomADODataSet* does not have an SQL property, but the descendant class *TADOQuery* does. So, to access the SQL property of the dataset referenced through the *DataSets* property, typecast it as a *TADOQuery*.

```
with (ADOConnection1.DataSets[10] as TADOQuery do begin
  SQL.Clear;
  SQL.Add('SELECT * FROM Species');
  Open;
end;
```

The *DataSetCount* property provides a total count of all of the datasets associated with a connection component. You can use the *DataSetCount* property as the basis for a loop with the *DataSets* property to sequentially visit all of the dataset components associated with a connection.

```
var
  i: Integer
begin
  for i := 0 to (ADOConnection4.DataSetCount) do
    ADOConnection4.DataSets[i].Open;
end;
```

Accessing the connection's commands

Topic groups See also

The *Commands* and *CommandCount* properties of *TADOConnection* act in much the same manner as the *DataSets* and *DataSetCount* properties. The difference is that *Commands* and *CommandCount* provide references to all of the *TADOCommand* components associated with the connection component. To work with all of the connection's dataset components, use the *DataSets* and *DataSetCount* properties.

Commands is a zero-based array of references to ADO command components. Use an index with *Commands* representing the position within the array of a particular command. For instance, use an index of 1 to reference the second command component in *Commands*.

```
Memo1.Lines.Text := ADOConnection1.Commands[1].CommandText;
```

The *CommandCount* property provides a total count of all of the commands associated with a connection component. You can use the *CommandCount* property as the basis for a loop with the *Commands* property to sequentially visit all of the command components associated with a connection.

```
var
  i: Integer
begin
  for i := 0 to (ADOConnection1.CommandCount) do
    ADOConnection1.Commands[i].Execute;
end;
```

Listing available tables

Topic groups See also

To get a listing of all of the tables contained in the database accessed via the connection object, use the *GetTableNames* method. This method copies a list of table names to an already-existing string list object. Use individual elements from this list for such things as the value for the *TableName* property of a *TADOTable* component or the name of a table in an SQL statement executed by a *TADOQuery*.

```
ADOConnection1.GetTableNames(ListBox1.Items, False);
```

The example below traverse a list of table names created using the *GetTableNames* method. For each table, the routine makes an entry in another table with the table's name and number of records.

```
procedure TForm1.Button1Click(Sender: TObject);
var
  SL: TStrings;
  index: Integer;
begin
  SL := TStringList.Create;
  try
    ADOConnection1.GetTableNames(SL, False);
    for index := 0 to (SL.Count - 1) do begin
      Table1.Insert;
      Table1.FieldByName('Name').AsString := SL[index];
      ADOTable1.TableName := SL[index];
      ADOTable1.Open;
      Table1.FieldByName('Records').AsInteger :=
        ADOTable1.RecordCount;
      Table1.Post;
    end;
  finally
    SL.Free;
    ADOTable1.Close;
  end;
end;
```

Listing available stored procedures

Topic groups See also

To get a listing of all of the stored procedures contained in the database accessed via the connection object, use the *GetProcedureNames* method. This method copies a list of stored procedure names to an already-existing string list object. One of the elements in the resulting list can be used for such things as the value for the *ProcedureName* property of a *TADOStoredProc* component.

```
ADOConnection1.GetProcedureNames(ListBox1.Items);
```

In the example below, a list of stored procedure names retrieved with *GetProcedureNames* is used to execute all of the stored procedures from the associated database.

```
procedure TDataForm.ExecuteProcsButtonClick(Sender: TObject);
var
  SL: TStrings;
  index: Integer;
begin
  SL := TStringList.Create;
  try
    ADOConnection1.GetProcedureNames(SL);
    if (SL.Count > 0) then
      for index := 0 to (SL.Count - 1) do begin
        ADOStoredProc1.ProcedureName := SL[index];
        ADOStoredProc1.ExecProc;
      end;
  finally
    SL.Free;
  end;
end;
```

Working with (connection) transactions

Topic groups See also

The *TADOConnection* component includes a number of methods and events for working with transactions. These transaction capabilities are shared by all of the ADO command and dataset components using the data store connection. Working with a connection component's transactions is covered in the topics:

- Using transaction methods
- Using transaction events

Using transaction methods

Topic groups See also

Use the methods *BeginTrans*, *CommitTrans*, and *RollbackTrans* to perform transaction processing. *BeginTrans* starts a transaction in the data store associated with the ADO connection component. *CommitTrans* commits a currently active transaction, saving changes to the database and ending the transaction. *RollbackTrans* cancels a currently active transaction, abandoning all changes made during the transaction and ending the transaction. Read the *InTransaction* property to determine at any given point whether the connection component has a transaction open.

A transaction started by the connection component is shared by all command and dataset components that use the connection established by the *TADOConnection* component.

Using transaction events

Topic groups See also

The ADO connection component provides a number of events for detecting when transaction-related processes have been completed. These events indicate when a transaction process initiated by a *BeginTrans*, *CommitTrans*, and *RollbackTrans* method have been successfully completed at the data store.

The *OnBeginTransComplete* event is triggered when the data store has successfully started a transaction after a call to the connection component's *BeginTrans* method. The *OnCommitTransComplete* event is triggered after a transaction is successfully committed due to a call to *CommitTrans*. And *OnRollbackTransComplete* is triggered after a transaction is successfully committed due to a call to *RollbackTrans*.

Using ADO datasets

Topic groups See also

The ADO dataset components provided in Delphi are analogous to the BDE-based dataset components. For instance, the *TADOTable* component is functionally equivalent to the *TTable* component. The main difference is that the ADO dataset components use underlying ADO objects for their data access and are not dependent on the Borland Database Engine for this.

The ADO dataset and BDE-based components have the *TDataSet* class as a common ancestor. Because of this, they share a common functionality in inherited or similar properties, methods, and events. This section primarily discusses areas of the ADO dataset components that differ from the corresponding generic dataset components. For more information on functionality common between the two sets of dataset components, see the descriptions for the generic and BDE-based dataset components:

- Understanding datasets
- Working with tables
- Working with queries
- Working with stored procedures

This section contains information pertaining to functionality that differs in the ADO versions of the dataset components from their generic counterparts. This information is divided into the areas:

- Features common to all ADO dataset components
- Using TADODataSet
- Using TADOTable
- Using TADOQuery
- Using TADOStoredProc

Features common to all ADO dataset components

Topic groups See also

Certain aspects of the ADO dataset components function exactly the same in all of the different components. Except as cited, these functional areas are used in exactly the same manner no matter which ADO dataset component is in use. The descriptions for these shared functional areas are consolidated in the sections:

- Modifying data
- Navigating in a dataset
- Using visual data-aware controls
- Connecting to a data store using ADO dataset components
- Working with record sets
- Using batch updates
- Loading data from and saving data to files
- Using parameters in commands

Modifying data

Topic groups See also

Accessing columns in ADO dataset components and modifying data is done in the exact same manner as in the generic dataset components. Use dataset methods like *Edit* and *Insert* to put the dataset in edit mode prior to changing data. Use the *Post* method to finalize data changes.

Use the dynamic *TField* references provided by the *Fields* property and *FieldByname* method of the dataset components. From there use the properties and methods of the *TField* class and descendants to do such things as setting or getting a column's value, validating data, and determining a column's data type.

For information on modifying data through dataset components, see Modifying data. For information on using table columns and persistent field objects, see Working with field components.

Navigating in a dataset

Topic groups See also

Navigating between rows in an ADO dataset is done in the same way as in generic dataset components. Use methods like *First*, *Next*, *Last*, and *Prior* to move the record pointer in the dataset component from one table row to another. Loops can be based on the *Eof* and *Bof* properties so that they operate on all of the rows that make up a dataset.

```
ADOTable1.First;
while not ADOTable1.Eof do begin
  { Process each record here }
  ...
  ADOTable1.Next;
end;
```

For information on navigating between table rows in dataset components, see Navigating datasets.

Using visual data-aware controls

Topic groups See also

The dataset provided by an ADO dataset component can be made available in an application using data-aware controls. Such datasets include the rows returned by a *TADOTable* component, the result set returned by a SELECT statement in a *TADOQuery*, and stored procedures that return a result set executed from a *TADOStoredProc* component.

To make these datasets available in data-aware controls:

1. Use a standard *TDataSource* component.
2. Specify an active ADO dataset component in its *DataSet* property.
3. Use the standard data-aware controls, like *TDBEdit* and *TDBGrid*.
4. Specify the *TDataSource* in the *DataSource* property of the data-aware control.

For example, creating this relationship between ADO dataset component, datasource component, and data-aware control programmatically:

```
DBGrid1.DataSource := DataSource1;
DataSource1.DataSet := ADOQuery1;
ADOQuery1.Open;
```

Connecting to a data store

Topic groups See also

ADO dataset components can connect to an ADO data store either collectively or individually.

When connecting dataset components collectively, set the *Connection* property of each dataset component to a *TADOConnection* component. Each dataset component then uses the connection established by that connection component.

```
ADODataSet1.Connection := ADOConnection1;
ADODataSet2.Connection := ADOConnection1;
...
```

Among the advantages of connecting dataset components collectively are:

- The dataset components share the connection object's attributes.
- Only one connection need be set up: that of the *TADOConnection*.
- The dataset components can participate in transactions.

When connecting dataset components individually, set the *ConnectionString* property of each dataset component. The information needed to connect to the data store must be set for each dataset component. Each dataset component establishes its own connection to the data store, totally independent of any other dataset connection in the application.

```
ADODataSet1.ConnectionString := 'Provider=YourProvider;Password=SecretWord;' +
  'User ID=JaneDoe;SERVER=PURGATORY;UID=JaneDoe;PWD=SecretWord;' +
  'Initial Catalog=Employee';
ADODataSet2.ConnectionString := 'Provider=YourProvider;Password=SecretWord;' +
  'User ID=JaneDoe;SERVER=PURGATORY;UID=JaneDoe;PWD=SecretWord;' +
  'Initial Catalog=Employee';
...
```

For more information on using a TADOConnection to connect to a data store see Connecting to a data store using TADOConnection.

Working with recordsets

Topic groups See also

In addition to the means for navigating between records and modifying data shared by all ADO dataset components, there are a number of other properties and methods for operating on record sets.

The *RecordSet* property provides direct access to the ADO recordset object underlying the dataset component. Using this object, it is possible to access properties and call methods of the recordset object from an application. Use of *RecordSet* to directly access the underlying ADO recordset object requires a good working knowledge of ADO objects in general and the ADO recordset object in specific. It is not recommended that you use the recordset object directly unless familiar with recordset object operations. Consult the Microsoft Data Access SDK help for specific information on using ADO recordset objects.

Use the *RecordSetState* property to determine the current state of the dataset component. *RecordSetState* implements the *State* property of the ADO recordset object, and so reflects the current state of the underlying recordset object. The *RecordSetState* property will contain one of the values: *stExecuting* or *stFetching*. A value of *stExecuting* indicates the dataset component is currently in the process of executing a command. A value of *stFetching* indicates the dataset component is in the process of fetching rows from the associated table (or tables).

Use these values to perform actions dependent on the current state of the dataset. For example, a routine that updates data might check the *RecordSetState* property to see whether the dataset is active and not in the process of other activities such as connecting or fetching data.

Using batch updates

Topic groups See also

ADO dataset components provide the ability to cache changes to the dataset and then either apply all of the changes as a batch operation or to cancel one or all of the changes. Batch updates can serve as a sort of transaction control, but at the dataset component level. (Ordinarily, transactions are handled as methods of the ADO connection component.)

Using the batch updates features of ADO dataset components is a matter of:

- Opening the dataset in batch update mode
- Inspecting the update status of individual rows
- Filtering multiple rows based on update status
- Applying the batch updates to base tables
- Canceling batch updates

Opening the dataset in batch update mode

Topic groups See also

To open an ADO dataset in batch update mode, it must meet these criteria:

1. The component's *CursorType* property must be *ctKeySet* (the default property value) or *ctStatic*.
2. The *LockType* property must be *ltBatchOptimistic*.
3. The command must be a SELECT query.

Before activating the dataset component, set the *CursorType* and *LockType* properties to the values indicated above. Assign a SELECT statement to the component's *CommandText* property (for *TADODataSet*) or the SQL property (for *TADOQuery*). For *TADOStoredProc* components, set the *ProcedureName* to the name of a stored procedure that returns a result set. These properties can be set at design-time through the Object Inspector or programmatically at runtime. The example below shows the preparation of a *TADODataSet* component for batch update mode.

```
with ADODataSet1 do begin
  CursorLocation := clUseServer;
  CursorType := ctKeyset;
  LockType := ltBatchOptimistic;
  CommandType := cmdText;
  CommandText := 'SELECT * FROM Employee';
  Open;
end;
```

After a dataset has been opened in batch update mode, all changes to the data are cached rather than applied directly to the base tables.

Inspecting the update status of individual rows

Topic groups See also

Determine the update status of a given row by making it current and then inspecting the *RecordStatus* property of the ADO data component. *RecordStatus* reflects the update status of the current row and only that row.

```
case ADOQuery1.RecordStatus of
  rsUnmodified: StatusBar1.Panels[0].Text := 'Unchanged record';
  rsModified:   StatusBar1.Panels[0].Text := 'Changed record';
  rsDeleted:    StatusBar1.Panels[0].Text := 'Deleted record';
  rsNew:        StatusBar1.Panels[0].Text := 'New record';
end;
```

Filtering multiple rows based on update status

Topic groups See also

Filter a recordset to show only those rows that belong to a group of rows with the same update status using the *FilterGroup* property. Set *FilterGroup* to the *TFilterGroup* constant that represents the update status of rows to display. A value of *fgNone* (the default value for this property) specifies that no filtering is applied and all rows are visible regardless of update status (except rows marked for deletion). The example below causes only pending batch update rows to be visible.

```
FilterGroup := fgPendingRecords;
Filtered := True;
```

For the *FilterGroup* property to have an effect, the ADO dataset component's *Filtered* property must be set to True.

Applying the batch updates to base tables

Topic groups See also

Apply pending data changes that have not yet been applied or canceled by calling the *UpdateBatch* method. Rows that have been changed and are applied have their changes put into the base tables on which the recordset is based. A cached row marked for deletion causes the corresponding base table row to be deleted. A record insertion (exists in the cache but not the base table) is added to the base table. Modified rows cause the columns in the corresponding rows in the base tables to be changed to the new column values in the cache.

Used alone with no parameter, *UpdateBatch* applies all pending updates. A *TUpdateBatchOptions* value can optionally be passed as the parameter for *UpdateBatch*. If any value except *ubAffectAll* is passed, only a subset of the pending changes are applied. Passing *ubAffectAll* is the same as passing no parameter at all and causes all pending updates to be applied. The example below applies only the currently active row to be applied:

```
ADODataSet1.UpdateBatch(ubAffectCurrent);
```

Canceling batch updates

Topic groups See also

Cancel pending data changes that have not yet been canceled or applied by calling the *CancelBatch* method. Rows that have been changed and are canceled have their columns values reverted back to the values that existed prior to the last call to *CancelBatch* or *UpdateBatch*, if either has been called, or prior to the current pending batch of changes.

Used alone with no parameter, *CancelBatch* cancels all pending updates. A *TUpdateBatchOptions* value can optionally be passed as the parameter for *CancelBatch*. If any value except *ubAffectAll* is passed, only a subset of the pending changes are canceled. Passing *ubAffectAll* is the same as passing no parameter at all and causes all pending updates to be canceled. The example below cancels all pending changes:

```
ADODataSet1.Cancel;
```

Loading data from and saving data to files

Topic groups See also

The data retrieved via an ADO dataset component can be saved to a file for later retrieval on the same or a different computer. Save the data to a file using the *SaveToFile* method. Retrieve the data from file using the *LoadFromFile* method. The data is saved in one of two proprietary formats: ADTG and XML. Indicate which of these two formats to use for the save file with one of the *TPersistFormat* constants *pfADTG* or *pfXML* in the *Format* parameter of the *SaveToFile* method.

In the example below, the first procedure saves the dataset retrieved by the *TADODataSet* component *ADODataSet1* to a file. The target file is an ADTG file named SaveFile, saved to a local drive. The second procedure loads this saved file into the *TADODataSet* component *ADODataSet2*.

```
procedure TForm1.SaveBtnClick(Sender: TObject);
begin
  if (FileExists('c:\SaveFile')) then begin
    DeleteFile('c:\SaveFile');
    StatusBar1.Panels[0].Text := 'Save file deleted!';
  end;
  ADODataSet1.SaveToFile('c:\SaveFile', pfADTG);
end;
procedure TForm1.LoadBtnClick(Sender: TObject);
begin
  if (FileExists('c:\SaveFile')) then
    ADODataSet2.LoadFromFile('c:\SaveFile')
  else
    StatusBar1.Panels[0].Text := 'Save file does not exist!';
end;
```

The saving and loading dataset components need not be on the same form as above, in the same application, or even on the same computer. This allows for the briefcase-style transfer of data from one computer to another.

On calling the *LoadFromFile* method, the dataset component is automatically activated.

If the file specified in the *FileName* parameter of the *SaveToFile* method already exists, an *EOleException* exception is raised. Similarly, if the file specified in the *FileName* parameter of *LoadFromFile* does not exist, an *EOleException* exception is raised.

The two save file formats ADTG and XML are the only formats supported by ADO. Even so, both formats are not necessarily supported in all versions of ADO. Consult the ADO documentation for the actual version in use to determine what save file formats are supported.

Using parameters in commands

Topic groups See also

Using parameters in commands and SQL statements executed as commands using ADO dataset components requires that you:

1. Include parameters in the SQL statement (identified by the prefixing colon).
2. Set the property values for each *TParameter* component.

For each token in the SQL statement identified as a parameter, one *TParameter* component is automatically created and added to the dataset component's *Parameters* property (a *TParameters* array of *TParameter* components). At design-time, access the parameter components to set their values using the property editor for the *Parameters* property. To invoke the property editor, click the ellipsis button for the *Parameters* property in the Object Inspector.

At runtime, access parameter components to set or get their values using the *Parameters* property of the dataset component. Specify an index number with *Parameters* that is the ordinal position of a specific parameter in the SQL statement (relative to other parameters). This index is zero based, so the first parameter is referenced with an index of zero, the second with an index of one, and so on. Alternately, use the *TParameters* reference provided by the *Parameters* property and call its *ParamByName* method to refer to the parameter by its name.

```
{ reference the first parameter with an index }
ADOQuery1.Parameters[0].Value := 'telephone';
{ reference a parameter by its name }
ADOQuery1.Parameters.ParamByName('Amount').Value := 123;
```

In the example below, the following SQL statement is used in a *TADOQuery* component.

```
SELECT CustNo, Company, State
FROM CUSTOMER
WHERE (State = :StateParam)
```

This statement has one parameter: *StateParam*. The routine below closes the ADO query component, sets the value of the *StateParam* parameter through the *Parameters* property, and then reopens the ADO query component. The *Parameters* property requires a parameter be identified by a number representing the parameter's ordinal position in the statement, relative to other parameters. *Parameters* is zero-based, so the first parameter is identified with a *Parameters* property index of zero, the second with a one, and so on. As *StateParam* is the first parameter in the statement, an index of zero is used to identify it.

```
procedure TForm1.GetCaliforniaBtnClick(Sender: TObject);
begin
  with ADOQuery1 do begin
    Close;
    Parameters[0].Value := 'CA';
    Open;
  end;
end;
```

The procedure below performs essentially the same purpose, but uses the *TParameters.ParamByName* method to set the parameter's value. The *ParamByName* method requires a parameter be identified by its name as used in the SQL statement (sans the colon).

```
procedure TForm1.GetFloridaBtnClick(Sender: TObject);
begin
  with ADOQuery1 do begin
    Close;
    Parameters.ParamByName('StateParam').Value := 'FL';
    Open;
  end;
end;
```

Using TADODataSet

Topic groups See also

The *TADODataSet* component provides Delphi applications the ability to access data from one or multiple tables in a database accessed via ADO. Tables accessed are specified using the *CommandText* property of the ADO dataset component, either by name or using an SQL statement.

The database is accessed using a data store connection established by the ADO dataset component using its *ConnectionString* property or through a separate *TADOConnection* component specified in the *Connection* property.

Data from a *TADODataSet* component is made available to an application in the same manner as the standard, BDE-centric data components. Use the ADO dataset component for the *DataSet* property of a standard *TDataSource* component. The *TDataSource* then acts as the data conduit between the ADO dataset component and data-aware controls.

Navigating in a *TADODataSet* component is also the same as in the standard data components. Use such methods inherited from the ancestor *TDataSet* as *First*, *Next*, *Last*, and *Prior* to programmatically move the row pointer through the dataset.

Programmatically modifying data is the same, too. Inherited methods like *Insert* and *Edit* put the *TADODataSet* in edit mode. *Post* finalizes data changes.

- Connecting to a data store using ADO dataset components
- Retrieving a dataset using a command
- Using visual data-aware controls
- Navigating in a dataset
- Modifying data
- Features common to all ADO dataset components

Retrieving a dataset using a command

Topic groups See also

The *TADODataSet* component is capable of retrieving data from a single table using the name of a table. It can also retrieve data from one or multiple tables using a valid SQL statement. In either case, the table name or SQL statement is executed as a command.

Specify the name of a table or an SQL statement in the *CommandText* property and activate the component. At design-time, you can use the Command Text Editor to build the command. To invoke this editor, click the ellipsis button in the *CommandText* property in the Object Inspector. At runtime, assign a command to *CommandText* as a **String**.

```
ADODataSet1.CommandText := 'SELECT * FROM Customer';
```

Use the *CommandType* property to indicate the type of command being executed: *cmdTable* (or *cmdTableDirect*) if the command is a table name or *cmdText* for SQL statements. You can also specify *cmdUnknown* if the command type is not known at time or execution or you wish ADO to make a guess at the command type based on the contents of *CommandText*. At design-time, select the desired value for *CommandType* from the drop-down list in the Object Inspector. At runtime, assign a value of type *TCommandType*.

```
ADODataSet1.CommandType := cmdText;
```

Activate the *TADODataSet* by calling its *Open* method or by assigning a value of True to the *Active* property.

```
with ADODataSet1 do begin
  Connection := ADOConnection1;
  CommandType := cmdText;
  CommandText := 'SELECT * FROM Customer';
  Open;
end;
```

Using TADOTable

Topic groups See also

The *TADOTable* component provides Delphi applications the ability to access data from a single table in a database accessed via ADO. The table accessed is specified in the *TableName* property of the ADO table component.

The database is accessed using a data store connection established by the ADO table component using its *ConnectionString* property or through a separate *TADOConnection* component specified in the *Connection* property.

Data from a *TADOTable* component is made available to an application in the same manner as the standard, BDE-centric data components. Use the ADO table component for the *DataSet* property of a standard *TDataSource* component. The *TDataSource* then acts as the data conduit between the ADO table component and data-aware controls.

Navigating in a *TADOTable* component is also the same as in the standard data components. Use such methods inherited from the ancestor *TDataSet* as *First*, *Next*, *Last*, and *Prior* to programmatically move the row pointer through the dataset.

Programmatically modifying data is the same, too. Inherited methods like *Insert* and *Edit* put the *TADOTable* in edit mode. *Post* finalizes data changes.

- Connecting to a data store using ADO dataset components
- Specifying the table to use
- Using visual data-aware controls
- Navigating in a dataset
- Modifying data
- Features common to all ADO dataset components

Specifying the table to use

Topic groups See also

Once a *TADOTable* component has a valid connection to a database, it can access tables contained in that database. Specify a single table of the database in the *TableName* property. When the ADO table component is activated, the table and its data become accessible through the *TADOTable*.

At design-time, if the *TADOTable* component has a valid data store connection, the property editor for the *TableName* property lists the names of available tables. Select one table from this list. At runtime, assign a **String** value containing a table name to the *TableName* property.

```
ADOTable1.TableName := 'Orders';
```

If a *TADOConnection* component is used to connect to the data store, you can use its *GetTableNames* method to retrieve a list of available tables. *GetTableNames* fills an already-existing string list object with the names of the tables available through the connection.

For example, the first routine below fills a *TListBox* component named *ListBox1* with the names of tables available through the *TADOConnection* component *ADOConnection1*. The second routine is a handler for the *OnDblClick* event of *ListBox1*. In this event handler, the currently selected table name in *ListBox1* is assigned to the *TableName* property of the *TADOTable* called *ADOTable1*. The ADO table component is then activated.

```
procedure TForm1.ListTablesButtonClick(Sender: TObject);
begin
  ADOConnection1.GetTableNames(ListBox1.Items, False);
end;
procedure TForm1.ListBox1DblClick(Sender: TObject);
begin
  with ADOTable1 do begin
    Close;
    TableName := (Sender as TListBox).Items[(Sender as TListBox).ItemIndex];
    Open;
  end;
end;
```

Using TADOQuery

Topic groups See also

The *TADOQuery* component provides Delphi applications the ability to access data from one or multiple tables from an ADO database using SQL. Specify the SQL statement to use with the ADO query component in the SQL property. *TADOQuery* can either retrieve data using data manipulation language (DML) or create and delete metadata objects using data definition language (DDL). The SQL used in a *TADOQuery* component must be acceptable to the ADO driver in use. Delphi performs no evaluation of the SQL and does not execute it. The SQL statement is merely passed to the database back-end for execution. If the SQL statement produces a result set, it is passed from the database back-end through Delphi to the *TADOQuery* for use by the application.

The database is accessed using a data store connection established by the ADO query component using its *ConnectionString* property or through a separate *TADOConnection* component specified in the *Connection* property.

Data from a *TADOQuery* component is made available to an application in the same manner as the standard, BDE-centric data components. Use the ADO query component for the *DataSet* property of a standard *TDataSource* component. The *TDataSource* then acts as the data conduit between the ADO query component and data-aware controls.

Navigating in a *TADOQuery* component is also the same as in the standard data components. Use such methods inherited from the ancestor *TDataSet* as *First*, *Next*, *Last*, and *Prior* to programmatically move the row pointer through the query result set.

Programmatically modifying data is the same, too. Inherited methods like *Insert* and *Edit* put the *TADOQuery* in edit mode. *Post* finalizes data changes.

- Connecting to a data store using ADO dataset components
- Specifying SQL statements
- Executing SQL statements
- Using parameters in commands
- Using visual data-aware controls
- Navigating in a dataset
- Modifying data
- Features common to all ADO dataset components

Specifying SQL statements

Topic groups See also

At design-time, invoke the property editor for the SQL property by clicking the ellipsis button in the Object Inspector. In the editor dialog, enter the SQL statement for the *TADOQuery*.

At runtime, assign a value to the SQL property. As is the case with the standard querying component, *TQuery*, the *TADOQuery.SQL* property is a string list object. Use properties and methods of the string list class to assign values to the SQL property.

In the example below, a SELECT statement is assigned to the SQL property of a *TADOQuery* component named *ADOQuery1*.

```
with ADOQuery1 do begin
  Close;
  with SQL do begin
    Clear;
    Add('SELECT Company, State');
    Add('FROM CUSTOMER');
    Add('WHERE State = ' + QuotedStr('HI'));
    Add('ORDER BY Company');
  end;
  Open;
end;
```

Executing SQL statements

Topic groups See also

A *TADOQuery* with a valid SQL statement in its SQL property can be executed in one of two ways. Which way is you use is predicated on the type of SQL statement the ADO query component is to execute.

If the SQL statement is one that returns a result set, the ADO query component should be activated by calling its *Open* method or by settings its *Active* property to True. Only SELECT statements return result sets, so a *TADOQuery* with a SELECT statement in its SQL property will always be activated using this approach.

```
ADOQuery1.SQL.Text := 'SELECT * FROM TrafficViolations';
ADOQuery1.Open;
```

Note that because methods cannot be called while designing an application in the Delphi IDE, only the *Active* property can be used to activate this kind of query at design-time. This is functionally the same as calling the *Open* method (at runtime).

Execute an SQL statement that does not return a result set by calling the *ExecSQL* method of the *TADOQuery* component. All SQL statements except SELECT fall into this category: INSERT, DELETE, UPDATE, CREATE INDEX, ALTER TABLE, and so on. A *TADOQuery* component with on of these SQL statements in its SQL property will always be activated using this approach.

```
ADOQuery1.SQL.Text := 'DELETE FROM TrafficViolations WHERE (TicketID = 1099)';
ADOQuery1.ExecSQL;
```

The *TADOCommand* component can be used to execute SQL statements like the one above that do not return result sets.

Using TADOStoredProc

Topic groups See also

The *TADOStoredProc* component provides Delphi applications the ability to execute stored procedures in a database accessed through an ADO data store. The stored procedure executed is specified in the *ProcedureName* property of the ADO stored procedure component.

The database is accessed using a data store connection established by the stored procedure component using its *ConnectionString* property or through a separate *TADOConnection* component specified in the *Connection* property.

Result sets retrieved by a *TADOStoredProc* component are made available to an application in the same manner as the standard, BDE-centric query component *TStoredProc*. Use the ADO stored procedure component for the *DataSet* property of a standard *TDataSource* component. The *TDataSource* then acts as the data conduit between the ADO stored procedure component and data-aware controls.

Data from a *TADOStoredProc* component is made available to an application in the same manner as the standard, BDE-centric data components. Use the ADO stored procedure component for the *DataSet* property of a standard *TDataSource* component. The *TDataSource* then acts as the data conduit between the ADO stored procedure component and data-aware controls.

Navigating in a *TADOStoredProc* component is also the same as in the standard data components. Use such methods inherited from the ancestor *TDataSet* as *First*, *Next*, *Last*, and *Prior* to programmatically move the row pointer through the result set returned by a stored procedure.

Programmatically modifying data is the same, too. Inherited methods like *Insert* and *Edit* put the *TADOStoredProc* in edit mode. *Post* finalizes data changes.

- Connecting to a data store using ADO dataset components
- Specifying the stored procedure
- Executing the stored procedure
- Using parameters with stored procedures
- Using visual data-aware controls
- Navigating in a dataset
- Modifying data
- Features common to all ADO dataset components

Specifying the stored procedure

Topic groups See also

Once a *TADOStoredProc* component has a valid connection to a database, it can execute stored procedures contained in that database. Specify the name of a stored procedure from the database in the *ProcedureName* property. Activate the ADO stored procedure component using its *Open* method (if it returns a result set) or its *ExecProc* method (if it does not).

At design-time, if the *TADOStoredProc* component has a valid data store connection, the property editor for the *ProcedureName* property lists the names of available stored procedures. Select a stored procedure from this list. At runtime, assign a **String** value containing a stored procedure name to the *ProcedureName* property.

```
ADOStoredProc1.ProcedureName := 'DeleteEmployee';
```

If a *TADOConnection* component is used to connect to the data store, you can use its *GetProcedureNames* method to retrieve a list of available stored procedures. *GetProcedureNames* fills an already-existing string list object with the names of the stored procedures available through the connection.

For example, the first routine below fills a *TListBox* component named *ListBox1* with the names of stored procedures available through the *TADOConnection* component *ADOConnection1*. The second routine is a handler for the *OnDblClick* event of *ListBox1*. In this event handler, the currently selected table name in *ListBox1* is assigned to the *ProcedureName* property of the *TADOStoredProc* called *ADOStoredProc1*. The ADO stored procedure component is then executed using the *ExecProc* method.

```
procedure TForm1.ListProceduresButtonClick(Sender: TObject);
begin
  ADOConnection1.GetProcedureNames(ListBox1.Items);
end;
procedure TForm1.ListBox1DblClick(Sender: TObject);
begin
  with ADOStoredProc1 do begin
    ProcedureName := TListBox(Sender).Items[TListBox(Sender).ItemIndex];
   ExecProc;
  end;
end;
```

Executing the stored procedure

Topic groups See also

A *TADOStoredProc* with the name of an existing stored procedure in its *ProcedureName* property can be executed in one of two ways. Which way is you use is predicated on whether or not the stored procedure returns a result set.

If the stored procedure is one that returns a result set, the ADO stored procedure component should be activated by calling its *Open* method or by settings its *Active* property to True.

```
ADOStoredProc1.ProcedureName := 'ShowPurebreds';
ADOStoredProc1.ExecProc;
```

Note that as methods cannot be called while designing an application in the Delphi IDE, only the *Active* property can be used to activate this kind of stored procedure at design-time.

Execute a stored procedure that does not return a result set by calling the *ExecProc* method of the *TADOStoredProc* component. All SQL statements except SELECT fall into this category: INSERT, DELETE, UPDATE, CREATE INDEX, ALTER TABLE, and so on. A *TADOQuery* component with on of these SQL statements in its SQL property will always be activated using this approach.

```
ADOStoredProc1.ProcedureName := 'DeletePoodles';
ADOStoredProc1.ExecProc;
```

Using parameters with stored procedures

Topic groups See also

The *TADOStoredProc* component is capable of accommodating three types of parameters (not all of which may be supported by all database types). A parameter may be for input, for output, or for returning a result set. This section describes using stored procedure parameters in these three roles. It is possible for a parameter to serve two purposes, such as being both an input and an output parameter. This is merely a variation on the three basic roles. Dual use of a parameter like this is a matter of combining two of the functional approaches described here.

The direction or purpose of a parameter is defined in the stored procedure when it is created. This direction cannot later be changed by a front-end application. For instance, you cannot use Delphi code to turn an input parameter into an output parameter. The stored procedure would need to be dropped and recreated, giving the parameter a new role in the process. The direction of a particular parameter is indicated in the *TParameter.Direction* property, which can be read either at design-time in the Object Inspector or programmatically at runtime.

Parameter Direction	Use
pdInput	Parameter used to supply a value to the stored procedure before execution.
pdOutput	Parameter used to return a singleton value from a stored procedure after execution.
pdInputOutput	Parameter that can be used as both an input and an output parameter, per the above definitions.
pdReturnValue	Parameter that contains a result set after execution.
pdUnknown	Parameter for which the direction could not be determined at the point of evaluation.

For more information on using each of these types of parameters with stored procedures, see:

- Using TADOStoredProc input parameters
- Using TADOStoredProc output parameters
- Using TADOStoredProc return value parameters

Using TADOStoredProc input parameters

Topic groups

Use an input parameter to supply a value to a stored procedure before executing that stored procedure. Such values are typically used in the WHERE clause of a stored procedure's SQL statement to limit the number of table rows affected. Assign the parameter a value before activating the *TADOStoredProc* by calling its *Open* method or setting it *Active* property to True (for stored procedures that return a result set) or before executing the *TADOStoredProc* with its *ExecProc* method.

At design-time, access parameters through the Object Inspector. With focus in the cell for the *Parameters* property, click the ellipsis button. This invokes the parameters editor dialog. Enter a value of the appropriate type in the *Value* property.

At runtime, assign a value to the *Value* property of the *TParameter* component for the target parameter. Use the *TParameter* reference provided by the *Parameters* property of the *TADOStoredProc*.

```
with ADOStoredProc1 do begin
  Close;
  Parameters[1].Value := 1;
  Open;
end;
```

Using TADOStoredProc output parameters

Topic groups See also

Use an input parameter to return a single value from a stored procedure. While a result set might consist of multiple rows of multiple columns, this output parameter must be the equivalent of one row containing one column. If the stored procedure uses a SELECT statement to retrieve this value, an attempt to return multiple values results in an exception.

An output parameter only contains a value after the stored procedure has been activated or executed. Note that not all database systems support returning both a result set and output parameters. Check the documentation for the particular database system you are using to verify what it supports in this regard. If a database system supports both returning a result set and output parameter values, the *TADOStoredProc* can be activated using either its *Open* method (or *Active* property) or its *ExecProc*. If the database system does not support both, you can only use output parameters to retrieve values by calling the *ExecProc* method.

At design-time, access parameters through the Object Inspector. With focus in the cell for the *Parameters* property, click the ellipsis button. This invokes the parameters editor dialog. If the *TADOStoredProc* is activated using its *Active* property (the only way to activate it at design-time), inspect the output parameter's *Value* property to see the value returned.

At runtime, assign a value to the *Value* property of the *TParameter* component for the target parameter. Use the *TParameter* reference provided by the *Parameters* property of the *TADOStoredProc*. For example, the routine below returns a **String** value through the output parameter named @OutParam1.

```
with ADOStoredProc1 do begin
  Close;
  ShowMessage(VarToStr(Parameters.ParamByName('@OutParam1').Value));
  ExecProc;
end;
```

Using TADOStoredProc return value parameters

Topic groups See also

Return value parameters need not be accessed directly. Instead, the result set returned through a return value parameter should be accessed as you would any dataset.

To make the result set available through visual data-aware controls, use a reference to the *TADOStoredProc* component as the value for the *DataSet* property of a *TDataSource* component. The *TDataSource* then acts as a conduit between the *TADOStoredProc* component and the data-aware controls. At design-time, this is done through the Object Inspector. In the *DataSet* property of the *TDataSource*, select the *TADOStoredProc* component from the drop-down list. At runtime, assign a reference to the *TADOStoredProc* to the *DataSet* property.

```
ADOStoredProc1.Close;
DataSource1.DataSet := ADOStoredProc1;
ADOStoredProc1.Open;
```

Alternately, the result set can be accessed and manipulated using navigation and editing properties and methods inherited from *TDataSet*. For information on modifying data through dataset components, see . For information on navigating between table rows in dataset components, see .

Executing commands

Topic groups See also

The set of ADO components provided in Delphi allows an application to execute commands. This section describes how to execute commands and what components to use to do so.

In the ADO environment, commands are textual representations of provider-specific action requests. Typically, they are Data Definition Language (DDL) and Data Manipulation Language (DML) SQL statements. The language used in commands is provider-specific, but usually compliant with the SQL-92 standard for the SQL language.

Commands can be executed from more than one Delphi component. Each command-capable component executes commands in a slightly different way, with varying strengths and weaknesses. Which component you should use for a particular command is predicated on the type of command and whether it returns a result set. In general, for commands that do not return a result set use the *TADOCommand* component (though the *TADOQuery* component can also execute these commands). For commands that do return a result set, either execute the command from a *TADODataSet* or use the command's statement in the SQL property of a *TADOQuery*.

The *TADOCommand* component provides the ability to execute commands, one command at a time. It is designed primarily for executing those commands that do not return result sets, such as Data Definition Language (DDL) SQL statements. Through an overloaded version of its Execute method, though, it is capable of returning a result set that can be used through an ADO dataset component.

Specify commands in the *CommandText* property. A command can optionally be described using the *CommandType* property. If no specific type is specified, the server is left to decide as best it can based on the command in *CommandText*. Commands used with an ADO command component can contain parameters for which values are substituted before execution of the command. Before a command can be executed, the ADO command component must have a valid connection to an ADO data store.

- Connecting to a data store using ADO dataset components
- Specifying the command
- Executing commands
- Canceling commands
- Retrieving result sets with commands
- Handling command parameters

Specifying the command

Topic groups See also

Specify the command to execute using the ADO command component in its CommandText property. At design-time, enter the command (an SQL statement, a table name, or the name of a procedure) in the *CommandText* property through the Object Inspector. At runtime, assign a **String** value containing the command to the *CommandText* property.

If desired, explicitly define the type of command being executed in the *CommandType* property. Among the constants for *CommandType* are: *cmdText* (used if the command is an SQL statement), *cmdTable* (if it is a table name), and *cmdStoredProc* (if the command is the name of a stored procedure). At design-time, select the appropriate command type from the list in the Object Inspector. At runtime, assign a value of type *TCommandType* to the *CommandType* property.

```
with ADOCommand1 do begin
  CommandText := 'AddEmployee';
  CommandType := cmdStoredProc;
...
end;
```

Using the Execute method

Topic groups See also

Before the command can be executed using an ADO command component, the *TADOCommand* must have a valid connection to a data store. See Connecting to a data store using ADO dataset components for more information this.

To execute the command call the *Execute* method of the ADO command component. For commands that do not require any parameters or execution options, call the simple overloaded version of *Execute* without any method parameters at all.

```
with ADOCommand1 do begin
  CommandText := 'UpdateInventory';
  CommandType := cmdStoredProc;
  Execute;
end;
```

For information on executing commands that return a result set, see Retrieving result sets with commands.

Canceling commands

Topic groups See also

After an attempt to execute a command has been initiated (with the *Execute* method of a *TADOCommand* component), it can be aborted by calling the *Cancel* method.

```
procedure TDataForm.ExecuteButtonClick(Sender: TObject);
begin
  ADOCommand1.Execute;
end;
procedure TDataForm.CancelButtonClick(Sender: TObject);
begin
  ADOCommand1.Cancel;
end;
```

The *Cancel* method only has an effect if there is a command pending and the command was executed asynchronously (*eoAsynchExecute* is in the *ExecuteOptions* parameter of the *Execute* method). A command is said to be pending if the *Execute* method has been called and the command has not yet been completed or timed out. If a command has not been aborted or completed before the number of seconds specified in the *CommandTimeout* property have expired, the command times out. If a timeout period of other than the default 30 seconds is desired, set the *CommandTimeout* property prior to calling the *Execute* method.

Retrieving result sets with commands

Topic groups See also

Before the command can be executed using an ADO command component, the *TADOCommand* must have a valid connection to a data store. See Connecting to a data store using ADO dataset components for more information this.

To execute a command returning a result set, call the *Execute* method of the ADO command component. The *Execute* method of *TADOCommand* returns an ADO recordset object. Assign this return value to the *RecordSet* property of an ADO dataset component such as a *TADODataSet*.

In the example below, the ADO record set produced by a call to the *Execute* method of a *TADOCommand* component (*ADOCommand1*) is assigned to the Recordset property of a *TADODataSet* component (*ADODataSet1*).

```
with ADOCommand1 do begin
  CommandText := 'SELECT Company, State ' +
    'FROM customer ' +
    'WHERE State = :StateParam';
  CommandType := cmdText;
  Parameters.ParamByName('StateParam').Value := 'HI';
  ADODataSet1.Recordset := Execute;
end;
```

As soon as this assignment is made to the ADO dataset component's *Recordset* property, the dataset component is activated (automatically) and the data is available. Use methods and properties of the dataset component to access the data programmatically. To make the data available using visual data-aware controls, use a *TDataSource* component as a conduit between the ADO dataset and the data-aware controls.

For information on executing commands that do not return a result set, see Executing commands.

Handling command parameters

Topic groups See also

Before the command can be executed using an ADO command component, the *TADOCommand* must have a valid connection to a data store. See Connecting to a data store using ADO dataset components for more information this.

Executing a command that has parameters is exactly the same as for those that do not, except that values must be assigned to the parameters before the command is executed.

For each parameter in the command, one *TParameter* object is automatically added to the *Parameters* property of the *TADOCommand* component. At design-time, use the Parameter Editor to access parameters, which is invoked by clicking the ellipsis button for the *Parameters* property in the Object Inspector. At runtime, use properties and methods of *TParameter* to set (or get) the values of each parameter.

```
with ADOCommand1 do begin
  CommandText := 'INSERT INTO Talley ' +
    '(Counter) ' +
    'VALUES (:NewValueParam)';
  CommandType := cmdText;
  Parameters.ParamByName('NewValueParam').Value := 57;
  Execute
end;
```

Access single *TParameter* objects in *Parameters* by a number representing the relative position (to each other) in the command using the *Parameters* property of the *TADOCommand* component. Reference the *TParameter* objects by their names using the *TParameters.ParamByName* method.

Creating and using a client dataset

Topic groups See also

TClientDataSet is a dataset component designed to work without the connectivity support of the Borland Database Engine (BDE) or ActiveX Data Objects (ADO). Instead, it uses MIDAS.DLL, which is much smaller and simpler to install and configure. You don't use database or ADO connection components with client datasets, because there is no database connection.

Client datasets provide all the data access, editing, navigation, data constraint, and filtering support introduced by *TDataset*. However, the application that uses a client dataset must provide the mechanism by which the client dataset reads data and writes updates. Client datasets provide for this in one of the following ways:

- Reading from and writing to a flat file accessed directly from a client dataset component. This is the mechanism used by flat-file database applications.
- Reading from another dataset. Client datasets provide a variety of mechanisms for copying data from other datasets. These are described in Copying data from another dataset.
- Using an *IAppServer* interface to obtain data from and post updates to a remote application server. This is the mechanism used by clients in a multi-tiered database application.

These mechanisms can be combined into a single application that employs the "briefcase model". Users take a snapshot of data, saving it to a flat-file so that they can work on it off-line. Later, the client application applies the changes from the local copy of data to the application server. The application server resolves them with the actual database, and returns errors to the client dataset for handling. For information building an application using the briefcase model, see Using the briefcase model.

The following topics provide more information about working with client datasets:

- Working with data using a client dataset describes the support introduced by TClientDataSet for manipulating the data it stores in memory.
- Using a client dataset with a data provider describes the properties, methods, and events that are specific to multi-tiered applications where a client dataset uses a provider component.
- Using a client dataset with flat-file data describes the methods that support loading, saving, and creating flat-file data for a client dataset.

Working with data using a client dataset

Topic groups See also

Like any dataset, you can use Client datasets to supply the data for data-aware controls using a data source component. See Using data controls for information on how to display database information in data-aware controls.

Because *TClientDataSet* is a descendant of *TDataSet*, client datasets inherit the power and usefulness of the properties, methods, and events defined for all dataset components. For a complete introduction to this generic dataset behavior, see Understanding datasets.

Client datasets differ from other datasets in that they hold all their data in memory. Because of this, their support for common database functions can involve additional capabilities or considerations. The following topics describe some of these common functions and the differences introduced by client datasets:

- Navigating data
- Limiting what records appear
- Representing master/detail relationships
- Constraining data values
- Making data read-only.
- Editing data .
- Sorting and indexing .
- Representing calculated values .
- Adding application-specific information to the data .

Navigating data in client datasets

Topic groups See also

If an application uses standard data-aware controls, then a user can navigate through a client dataset's records just as for any other dataset. You can also navigate programmatically through records using standard dataset methods such as *First*, *GotoKey*, *Last*, *Next*, and *Prior*. For more information about these methods, see Navigating datasets.

Client datasets also support standard bookmark capabilities for marking and navigating to specific records. For more information about bookmarking, see Marking and returning to records.

Unlike most datasets, client datasets can also position the cursor at any specific record in the dataset by using the *RecNo* property. Ordinarily an application uses *RecNo* to determine the record number of the current record. Client datasets can, however, set *RecNo* to a particular record number to make that record the current one.

Limiting what records appear

Topic groups See also

To restrict users to a subset of available data on a temporary basis, applications can use ranges and filters. When you apply a range or a filter, the client dataset does not display all the data in its in-memory cache. Instead, it only displays the data that meets the range or filter conditions. For more information about using filters, see Displaying and editing a subset of data using filters. For more information about ranges, see Working with a subset of data.

With most datasets, filter strings are parsed into SQL commands that are then implemented on the database server. Because of this, the SQL dialect of the server limits what operations are used in filter strings. Client datasets implement their own filter support, which includes more operations than with other datasets. For example, when using a client dataset, filter expressions can include string operators that return substrings, operators that parse date/time values, and much more. Client datasets also allow filters on BLOB fields or complex field types such as ADT fields and array fields. See *Filter* for details.

When applying ranges or filters, the client dataset still stores all of its records in memory. The range or filter merely determines which records are available to controls that navigate or display data from the client dataset. In multi-tiered applications, you can also limit the data that is stored in the client dataset to a subset of records by supplying parameters to the application server. For more information on this, see Limiting records with parameters.

Representing master/detail relationships

Topic groups See also

Like tables, client datasets support master/detail forms. When you set up a master/detail relationship, you link two datasets so that all the records of one (the detail) always correspond to the single current record in the other (the master). For more information about master/detail forms, see Creating master/detail forms.

In addition, you can set up master/detail relationships in client datasets using nested tables. You can do this in one of two ways:

- Obtain records that contain nested details from a provider component. When a provider component represents the master table of a master/detail relationship, it automatically creates a nested dataset field to represent the details for each record.
- Define nested details using the Fields Editor. At design time, right click the client dataset and choose Fields Editor. Add a new persistent field to your client dataset by right-clicking and choosing Add Fields. Define your new field with type DataSet Field. In the Fields Editor, define the structure of your detail table.

When your client dataset contains nested detail datasets, *TDBGrid* provides support for displaying the nested details in a popup window. Alternately, you can display and edit these datasets in data-aware controls by using a separate client dataset for the detail set. At design time, create persistent fields for the fields in your client dataset, including a DataSet field for the nested detail set.

You can now create a client dataset to represent the nested detail set. Set this detail client dataset's *DataSetField* property to the persistent DataSet field in the master dataset.

In multi-tiered applications, using nested detail sets is necessary if you want to apply updates from master and detail tables to the application server. In flat-file database applications, using nested detail sets lets you save the details with the master records in one flat-file, rather than requiring you load two datasets separately, and then recreate the indexes to re-establish the master/detail relationship.

Note: To use nested detail sets, the *ObjectView* property of the client dataset must be *True*.

Constraining data values

Topic groups See also

Client datasets provide support for data constraints. You can always supply custom constraints. This lets you provide your own, application-defined limits on what values users post to a client dataset. For more information about supplying custom constraints, see Adding custom constraints.

In addition, if you are using a provider component to communicate with a remote database server, the provider has the option of supplying server constraints to the client dataset. For more information about controlling whether the application server communicates data constraints to a client dataset, see Handling server constraints.

In multi-tiered applications, there may be times when you want to turn off enforcement of data constraints, especially when the client dataset does not contain all of the records from the corresponding dataset on the application server. See Handling constraints for more information on how and why to do this.

Making data read-only

Topic groups See also

TDataSet introduces the *CanModify* property so that applications can determine whether the data in a dataset can be edited. Applications can't change the *CanModify* property, because for some *TDataSet* descendants, the underlying database, not the application, controls whether data can be modified.

However, because client datasets represent in-memory data, your application can always control whether users can edit that data. To prevent users from modifying data, set the *ReadOnly* property of the client dataset to *True*. Setting *ReadOnly* to *True* sets the *CanModify* property to *False*.

Unlike other kinds of datasets, you do not need to close a client dataset to change its read-only status. An application can make a client dataset read-only or not on a temporary basis at any time merely by changing the current setting of the *ReadOnly* property.

Editing data

Topic groups See also

Client datasets represent their data as an in-memory data packet. This packet is the value of the client dataset's *Data* property. By default, however, edits are not stored in the *Data* property. Instead the insertions, deletions, and modifications (made by users or programmatically) are stored in an internal change log, represented by the *Delta* property. Using a change log serves two purposes:

- When working with a provider, the change log is required by the mechanism for applying updates to the application server.
- In any application, the change log provides sophisticated support for undoing changes.

The *LogChanges* property enables you to disable logging temporarily. When *LogChanges* is *True*, changes are recorded in the log. When *LogChanges* is *False*, changes are made directly to the *Data* property. You can disable the change log in single-tiered applications when you do not need the undo support.

Edits in the change log remain there until they are removed by the application. Applications remove edits when

- Undoing changes
- Saving changes

Note: Saving the client dataset to a file does not remove edits from the change log. When you reload the dataset, the *Data* and *Delta* properties are the same as they were when the data was saved.

Undoing changes

Topic groups See also

Even though a record's original version remains unchanged in *Data*, each time a user edits a record, leaves it, and returns to it, the user sees the last changed version of the record. If a user or application edits a record a number of times, each changed version of the record is stored in the change log as a separate entry.

Storing each change to a record makes it possible to support multiple levels of undo operations should it be necessary to restore a record's previous state:

- To remove the last change to a record, call *UndoLastChange*. *UndoLastChange* takes a Boolean parameter, *FollowChange*, that indicates whether or not to reposition the cursor on the restored record (*True*), or to leave the cursor on the current record (*False*). If there are several changes to a single record, each call to *UndoLastChange* removes another layer of edits. *UndoLastChange* returns a Boolean value indicating success or failure to remove a change. If the removal occurs, *UndoLastChange* returns *False*. Use the *ChangeCount* property to determine whether there are any more changes to undo. *ChangeCount* indicates the number of changes stored in the change log.
- Instead of removing each layer of changes to a single record, you can remove them all at once. To remove all changes to a record, select the record, and call *RevertRecord*. *RevertRecord* removes any changes to the current record from the change log.
- At any point during edits, you can save the current state of the change log using the *SavePoint* property. Reading *SavePoint* returns a marker into the current position in the change log. Later, if you want to undo all changes that occurred since you read the save point, set *SavePoint* to the value you read previously. Your application can obtain values for multiple save points. However, once you back up the change log to a save point, the values of all save points that your application read after that one are invalid.
- You can abandon all changes recorded in the change log by calling *CancelUpdates*. *CancelUpdates* clears the change log, effectively discarding all edits to all records. Be careful when you call *CancelUpdates*. After you call *CancelUpdates*, you cannot recover any changes that were in the log.

Saving changes

Topic groups See also

Client datasets use different mechanisms for incorporating changes from the change log, depending on whether they are used in a stand-alone application or represent data from a remote application server. Whichever mechanism is used, the change log is automatically emptied when all updates have been incorporated.

Stand-alone applications can simply merge the changes into the local cache represented by the *Data* property. They do not need to worry about resolving local edits with changes made by other users. To merge the change log into the *Data* property, call the *MergeChangeLog* method. Merging changes into data describes this process.

You can't use *MergeChangeLog* in multi-tiered applications. The application server needs the information in the change log so that it can resolve updated records with the data stored in the database. Instead, you call *ApplyUpdates*, which sends the changes to the application server and updates the *Data* property only when the modifications have been successfully posted to the database. See Applying updates for more information about this process.

Sorting and indexing

Topic groups See also

Using indexes provides several benefits to your applications:

- They allow client datasets to locate data quickly.
- They enable your application to set up relationships between client datasets such as lookup tables or master/detail forms.
- They specify the order in which records appear.

If a client dataset is used in a multi-tiered application, it inherits a default index and sort order based on the data it receives from the application server. The default index is called DEFAULT_ORDER. You can use this ordering, but you cannot change or delete the index.

In addition to the default index, the client dataset maintains a second index, called CHANGEINDEX, on the changed records stored in the change log (*Delta* property). CHANGEINDEX orders all records in the client dataset as they would appear if the changes specified in *Delta* were applied. CHANGEINDEX is based on the ordering inherited from DEFAULT_ORDER. As with DEFAULT_ORDER, you cannot change or delete the CHANGEINDEX index.

You can use other existing indexes for a dataset, and you can create your own indexes. The following sections describe how to create and use indexes with client datasets:

- Adding a new index
- Deleting and switching indexes
- Using indexes to group data
- Indexing on the fly

Adding a new index

Topic groups See also

To create a new index for a client dataset, call *AddIndex*. *AddIndex* lets you specify the properties of the index, including

- The name of the index. This can be used for switching indexes at runtime.
- The fields that make up the index. The index uses these fields to sort records and to locate records that have specific values on these fields.
- How the index sorts records. By default, indexes impose an ascending sort order (based on the machine's locale). This default sort order is case-sensitive. You can specify options to make the entire index case sensitive or to sort in descending order. Alternately, you can provide a list of fields that should be sorted case-insensitively and a list of fields that should be sorted in descending order.
- The default level of grouping support for the index.

Tip: You can index and sort on internally calculated fields with client datasets.

Indexes you create are sorted in ascending alphabetical order according to your machine's locale. In the index options parameter you can add *ixDescending* to the set of options to override this default.

Note: When you create indexes at the same time as the client dataset, you can create indexes that sort in ascending order on some fields and descending order on others. See Creating a dataset using field and index definitions for details.

Sorting of string fields in indexes is case sensitive by default. In the index options parameter you can add *ixCaseInsensitive* to ignore case when sorting.

Note: When you create indexes at the same time as the client dataset, you can create indexes that are case insensitive on some fields and case sensitive on others. See Creating a dataset using field and index definitions for details.

Warning: Indexes you add using *AddIndex* are not saved when you save the client dataset to a file.

Deleting and switching indexes

Topic groups See also

To remove an index you created for a client dataset, call *DeleteIndex* and specify the name of the index to remove. You cannot remove the DEFAULT_ORDER and CHANGEINDEX indexes.

To use a different index with a client dataset when more than one index is available, use the *IndexName* property to select the index to use. At design time, you can select from available indexes in *IndexName* property drop-down box in the Object Inspector.

Using indexes to group data

Topic groups See also

When you use an index in your client dataset, it automatically imposes a sort order on the records. Because of this order, adjacent records usually contain duplicate values on the fields that make up the index. For example, consider the following fragment from an orders table that is indexed on the SalesRep and Customer fields:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200
1	2	6	75
2	1	1	10
2	3	4	200

Because of the sort order, adjacent values in the SalesRep column are duplicated. Within the records for SalesRep 1, adjacent values in the Customer column are duplicated. That is, the data is grouped by SalesRep, and within the SalesRep group it is grouped by Customer. Each grouping has an associated level. In this case, the SalesRep group has level 1 (because it is not nested in any other groups) and the Customer group has level 2 (because it is nested in the group with level 1). Grouping level corresponds to the order of fields in the index. When you create an index, you can specify the level of grouping it supports (up to the number of fields in the index).

Client datasets allow you to determine where the current record lies within any given grouping level. This allows your application to display records differently, depending on whether they are the first record in the group, in the middle of a group, or the last record in a group. For example, you might want to only display a field value if it is on the first record of the group, eliminating the duplicate values. To do this with the previous table results in the following:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
		2	50
	2	3	200
		6	75
2	1	1	10
	3	4	200

To determine where the current record falls within any group, use the *GetGroupState* method. *GetGroupState* takes an integer giving the level of the group and returns a value indicating where the current record falls the group (first record, last record, or neither).

Indexing on the fly

Topic groups See also

Instead of creating an index that becomes part of the client dataset, you can create a temporary index on the fly by specifying fields to use for indexing in the *IndexFieldNames* property. Separate field names with semicolons. Ordering of field names in the list is significant.

Note: When indexing on the fly, you can't specify a descending or case-insensitive index.

Warning: Indexes created on the fly do not support grouping.

Representing calculated values

Topic groups See also

As with any dataset, you can add calculated fields to your client dataset. These are fields whose values you calculate dynamically, usually based on the values of other fields in the same record.

Client datasets, however, let you optimize when fields are calculated by using internally calculated fields.

You can also tell client datasets to create calculated values that summarize the data in several records using maintained aggregates.

Using internally calculated fields in client datasets

Topic groups See also

In other datasets, your application must compute the value of calculated fields every time the record changes or the user edits any fields in the current record. It does this in an *OnCalcFields* event handler.

While you can still do this, client datasets let you minimize the number of times calculated fields must be recomputed by saving calculated values in the client dataset's data. When calculated values are saved with the client dataset, they must still be recomputed when the user edits the current record, but your application need not recompute values every time the current record changes. To save calculated values in the client dataset's data, use internally calculated fields instead of calculated fields.

Internally calculated fields, just like calculated fields, are calculated in an *OnCalcFields* event handler. However, you can optimize your event handler by checking the *State* property of your client dataset. When *State* is *dsInternalCalc*, you must recompute internally calculated fields. When *State* is *dsCalcFields*, you need only recompute regular calculated fields.

To use internally calculated fields, you must define the fields as internally calculated before you create the client dataset. If you are creating the client dataset using persistent fields, define fields as internally calculated by selecting InternalCalc in the Fields editor. If you are creating the client dataset using field definitions, set the *InternalCalcField* property of the relevant field definition to *True*.

Note: Other types of datasets use internally calculated fields. However, with other datasets, you do not calculate these values in an *OnCalcFields* event handler. Instead, they are computed automatically by the BDE or remote database server.

Using maintained aggregates

Topic groups See also

Client datasets provide support for summarizing data over groups of records. Because these summaries are automatically updated as you edit the data in the dataset, this summarized data is called a “maintained aggregate.”

In their simplest form, maintained aggregates let you obtain information such as the sum of all values in a column of the client dataset. They are flexible enough, however, to support a variety of summary calculations and to provide subtotals over groups of records defined by a field in an index that supports grouping.

The following topics describe how to

- Specify aggregates.
- Aggregate over groups of records.
- Obtain aggregate values.

Specifying aggregates

Topic groups See also

To specify that you want to calculate summaries over the records in a client dataset, use the *Aggregates* property. *Aggregates* is a collection of aggregate specifications (*TAggregate*). You can add aggregate specifications to your client dataset using the Collection Editor at design time, or using the *Add* method of *Aggregates* at runtime. If you want to create field components for the aggregates, create persistent fields for the aggregated values in the Fields Editor.

Note: When you create aggregated fields, the appropriate aggregate objects are added to the client dataset's *Aggregates* property automatically. Do not add them explicitly when creating aggregated persistent fields.

For each aggregate, the *Expression* property indicates the summary calculation it represents. *Expression* can contain a simple summary expression such as

```
Sum(Field1)
```

or a complex expression that combines information from several fields, such as

```
Sum(Qty * Price) - Sum(AmountPaid)
```

Aggregate expressions include one or more of the summary operators in the following table

Operator	Use
Sum	Totals the values for a numeric field or expression
Avg	Computes the average value for a numeric or date-time field or expression
Count	Specifies the number of non-blank values for a field or expression
Min	Indicates the minimum value for a string, numeric, or date-time field or expression
Max	Indicates the maximum value for a string, numeric, or date-time field or expression

The summary operators act on field values or on expressions built from field values using the same operators you use to create filters. (You can't nest summary operators, however.) You can create expressions by using operators on summarized values with other summarized values, or on summarized values and constants. However, you can't combine summarized values with field values, because such expressions are ambiguous (there is no indication of which record should supply the field value.) These rules are illustrated in the following expressions:

Sum(Qty * Price)	{**legal** -- summary of an expression on fields }
Max(Field1) - Max(Field2)	{**legal** -- expression on summaries }
Avg(DiscountRate) * 100	{**legal** -- expression of summary and constant }
Min(Sum(Field1))	{**illegal** -- nested summaries }
Count(Field1) - Field2	{**illegal** -- expression of summary and field }

Aggregating over groups of records

Topic groups See also

By default, maintained aggregates are calculated so that they summarize all the records in a client dataset. However, you can specify that you want to summarize over the records in a group instead. This allows you to provide intermediate summaries such as subtotals for groups of records that share a common field value.

Before you can specify a maintained aggregate over a group of records, you must use an index that supports the appropriate grouping. Once you have an index that groups the data in the way you want it summarized, specify the *IndexName* and *GroupingLevel* properties of the aggregate to indicate what index it uses, and which group or subgroup on that index defines the records it summarizes.

For example, consider the following fragment from an orders table that is grouped by SalesRep and, within SalesRep, by Customer:

SalesRep	Customer	OrderNo	Amount
1	1	5	100
1	1	2	50
1	2	3	200
1	2	6	75
2	1	1	10
2	3	4	200

The following code sets up a maintained aggregate that indicates the total amount for each sales representative:

```
Agg.Expression := 'Sum(Amount)';
Agg.IndexName := 'SalesCust';
Agg.GroupingLevel := 1;
Agg.AggregateName := 'Total for Rep';
```

To add an aggregate that summarizes for each customer within a given sales representative, create a maintained aggregate with level 2.

Maintained aggregates that summarize over a group of records are associated with a specific index. The *Aggregates* property can include aggregates that use different indexes. However, only the aggregates that summarize over the entire dataset and those that use the current index are valid. Changing the current index changes which aggregates are valid. To determine which aggregates are valid at any time, use the *ActiveAggs* property.

Obtaining aggregate values

Topic groups See also

To get the value of a maintained aggregate, call the *Value* method of the *TAggregate* object that represents the aggregate. *Value* returns the maintained aggregate for the group that contains the current record of the client dataset.

When you are summarizing over the entire client dataset, you can call *Value* at any time to obtain the maintained aggregate. However, when you are summarizing over grouped information, you must be careful to ensure that the current record is in the group whose summary you want. Because of this, it is a good idea to obtain aggregate values at clearly specified times, such as when you move to the first record of a group or when you move to the last record of a group. Use the *GetGroupState* method to determine where the current record falls within a group.

To display maintained aggregates in data-aware controls, use the Fields editor to create a persistent aggregate field component. When you specify an aggregate field in the Fields editor, the client dataset's *Aggregates* is automatically updated to include the appropriate aggregate specification. The *AggFields* property contains the new aggregated field component, and the *FindField* method returns it.

Adding application-specific information to the data

Topic groups See also

Application developers can add custom information to the client dataset's *Data* property. Because this information is bundled with the data packet, it is included when you save the data to a file or stream. It is copied when you copy the data to another dataset. Optionally, it can be included with the *Delta* property so that an application server can read this information when it receives updates from the client dataset.

To save application-specific information with the *Data* property, use the *SetOptionalParam* method. This method lets you store an OleVariant that contains the data under a specific name.

To retrieve this application-specific information, use the *GetOptionalParam* method, passing in the name that was used when the information was stored.

Copying data from another dataset

Topic groups See also

To copy the data from another dataset at design time, right click the dataset and choose Assign Local Data. A dialog appears listing all the datasets available in your project. Select the one you want to copy from and choose OK. When you copy the source dataset, your client dataset is automatically activated.

To copy from another dataset at runtime, you can assign its data directly or, if the source is another client dataset, you can clone the cursor.

Assigning data directly

Topic groups See also

You can use the client dataset's *Data* property to assign data to a client dataset from another dataset. *Data* is an OleVariant in the form of a data packet. A data packet can come from another client dataset, or from any other dataset by using a provider. Once a data packet is assigned to *Data*, its contents are displayed automatically in data-aware controls connected to the client dataset by a data source component.

When you open a client dataset that uses a provider component, data packets are automatically assigned to *Data*. See Using a client dataset with a data provider for more information on using providers with client datasets.

When your client dataset does not use a provider, you can copy the data from another client dataset as follows:

```
ClientDataSet1.Data := ClientDataSet2.Data;
```

Note: When you copy the *Data* property of another client dataset, you copy the change log as well, but the copy does not reflect any filters or ranges that have been applied. To include filters or ranges, you must clone the source dataset's cursor instead.

If you are copying from a dataset other than a client dataset, you can create a dataset provider component, link it to the source dataset, and then copy its data:

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := SourceDataSet;
ClientDataSet1.Data := TempProvider.Data;
TempProvider.Free;
```

Note: When you assign directly to the *Data* property, the new data packet is not merged into the existing data. Instead, all previous data is replaced.

If you want to merge changes from another dataset, rather than copying its data, you must use a provider component. Create a dataset provider as in the previous example, but attach it to the destination dataset and instead of copying the data property, use the *ApplyUpdates* method:

```
TempProvider := TDataSetProvider.Create(Form1);
TempProvider.DataSet := ClientDataSet1;
TempProvider.ApplyUpdates(SourceDataSet.Delta, -1, ErrCount);
TempProvider.Free;
```

Cloning a client dataset cursor

Topic groups See also

TClientDataSet provides the *CloneCursor* procedure to enable you to work with a second view of a specified client dataset at runtime. *CloneCursor* lets a second client dataset share the original client dataset's data. This is less expensive than copying all the original data, but, because the data is shared, the second client dataset can't modify the data without affecting the original client dataset.

CloneCursor takes three parameters: *Source* specifies the client dataset to clone. The last two parameters (*Reset* and *KeepSettings*) indicate whether to copy information other than the data. This information includes any filters, the current index, links to a master table (when the source dataset is a detail set), the *ReadOnly* property, and any links to a connection component or provider interface.

When *Reset* and *KeepSettings* are *False*, a cloned client dataset is opened, and the settings of the source client dataset are used to set the properties of the destination. When *Reset* is *True*, the destination dataset's properties are given the default values (no index or filters, no master table, *ReadOnly* is *False*, and no connection component or provider is specified). When *KeepSettings* is *True*, the destination dataset's properties are not changed.

Using a client dataset with a data provider

Topic groups See also

When using a client dataset in a multi-tiered application, the client dataset obtains data from a provider on the application server and, after editing that data locally, applies updates to the remote database. It is also possible to use a client dataset with a provider that resides in the same application.

The following steps describe how to use a client dataset with a provider:

1. Specifying a data provider.
2. Optionally, Getting parameters from the application server or Passing parameters to the application server.
3. Optionally, Overriding the dataset on the application server.
4. Requesting data from an application server.
5. Handling constraints received from the application server.
6. Updating records.
7. Refreshing records.

In addition, client datasets allow you to communicate with the provider using a custom event.

Specifying a data provider

Topic groups See also

Before a client dataset can receive data from and apply updates to an application server, it must be associated with a dataset provider. To associate a client dataset with a provider in a multi-tiered application, use the *RemoteServer* and *ProviderName* properties. In single-tiered applications and in client applications used in briefcase mode as temporary single-tiered applications, these properties are not used. When using a client dataset with a provider that is instantiated in the same application, you do not need to use the *RemoteServer* property, but you can still use the *ProviderName* property, as long as the provider component has the same *Owner* as the client dataset.

RemoteServer specifies the name of a connection component from which to get a list of providers. The connection component resides in the same data module as the client dataset. It establishes and maintains a connection to an application server, sometimes called a “data broker”. For more information, see The structure of the client application

At design time, after you specify *RemoteServer*, you can select a provider from the drop-down list for the *ProviderName* property in the Object Inspector. This list also includes any local providers that belong to the same data module. At runtime, you can switch among available providers by setting *ProviderName* in code.

To use a local provider that has a different *Owner*, you must form the association at runtime using the client dataset’s *SetProvider* method.

Getting parameters from the application server

Topic groups See also

There are two circumstances when the client application needs to obtain parameter values from the application server:

- The client needs to know the value of output parameters on a stored procedure.
- The client wants to initialize the input parameters of a query or stored procedure to the current values of a query or stored procedure on the application server.

A client dataset stores parameter values in its *Params* property. These values are refreshed with any output parameters whenever the client dataset fetches data from the application server. However, there are times a client application needs output parameters when it is not fetching data.

To fetch output parameters when not fetching records, or to initialize input parameters, the client dataset can request parameter values from the application server by calling the *FetchParams* method. The parameters are returned in a data packet from the application server and assigned to the client dataset's *Params* property.

At design time, the *Params* property can be initialized by right-clicking the client dataset and choosing Fetch Params.

Note: The *FetchParams* method (or the Fetch Params command) will only work if the client dataset is connected to a provider whose associated dataset can supply parameters. *TDataSetProvider* can supply parameter values if it represents a query or stored procedure.

When working with a stateless application server, you can't use *FetchParams* to retrieve output parameters. In a stateless application server, other clients can change and rerun the query or stored procedure, changing output parameters before the call to *FetchParams*. To retrieve output parameters from a stateless application server, use the *Execute* method. If the provider is associated with a query or stored procedure, *Execute* tells the provider to execute the query or stored procedure and return any output parameters. These returned parameters are then used to automatically update the *Params* property.

The *Params* property can also be used to pass parameter values to the application server. For more information, see Passing parameters to the application server.

Passing parameters to the application server

Topic groups See also

Client datasets can pass parameters to the application server to specify what data they want provided in the data packets it sends. These parameters can specify

- Parameter values for a query or stored procedure that is run on the application server
- Field values that limit the records sent in data packets

You can specify parameter values that your client dataset sends to the application server at design time or at runtime. At design time, select the client dataset, and then double-click the *Params* property in the Object Inspector. This brings up the collection editor, where you can add, delete, or rearrange parameters. By selecting a parameter in the collection editor, you can use the Object Inspector to edit the properties of that parameter.

At runtime, use the *CreateParam* method of the *Params* property to add parameters to your client dataset. *CreateParam* returns a parameter object, given a specified name, parameter type, and datatype. You can then use the properties of that parameter object to assign a value to the parameter.

For example, the following code sets the value of a parameter named CustNo to 605:

```
with ClientDataSet1.Params.CreateParam(ftInteger, 'CustNo', ptInput) do
     AsInteger := 605;
```

If the client dataset is not active, you can send the parameters to the application server and retrieve a data packet that reflects those parameter values simply by setting the *Active* property to *True*.

Note: You may want to initialize parameter values from the current settings on the application server. You can do this by right-clicking the client dataset and choosing Fetch Params at design time or calling the *FetchParams* method at runtime.

Sending query or stored procedure parameters

Topic groups See also

When the provider on the application server represents the results of a query or stored procedure, you can use the *Params* property to specify parameter values. When the client dataset requests data from the application server or uses its *Execute* method to run a query or stored procedure that does not return a dataset, it passes these parameter values along with the request for data or the execute command. When the provider receives these parameter values, it assigns them to its associated query or stored procedure. The application server then runs the query or stored procedure using these parameter values, and, if the client dataset requested data, begins providing data, starting with the first record in the result set.

Note: Parameter names should match the names of the corresponding parameters on the query or stored procedure component in the application server.

Limiting records with parameters

Topic groups See also

When the provider on the application server represents the results of a table component, you can use *Params* property to limit the records that are provided to the *Data* property.

Each parameter name must match the name of a field in the *TTable* component on the application server. The provider component on the application server sends only those records whose values on the corresponding fields match the values assigned to the parameters.

For example, consider a client application that displays the orders for a single customer. When the user identifies the customer, the client dataset sets its *Params* property to include a single parameter named CustID (or whatever field in the server table is called) whose value identifies the customer whose orders it will display. When the client dataset requests data from the application server, it passes this parameter value. The application server then sends only the records for the identified customer. This is more efficient than letting the application server send all the orders records to the client application and then filtering the records on the client side.

Overriding the dataset on the application server

Topic groups See also

Usually, the provider on the application server is associated with a dataset that determines what data is supplied to clients. This dataset may have a property that specifies an SQL statement to generate the data, or it may represent a specific database table or stored procedure.

If the provider allows, the client dataset can override the property that indicates what data the dataset represents. To do so, you can set the client dataset's *CommandText* property. *CommandText* contains an SQL statement that replaces the SQL on the provider's dataset, or it contains the name of a table or stored procedure that replaces the table or stored procedure that the dataset currently represents. This allows the client dataset to specify dynamically what data it wants to see.

By default, provider's do not allow client datasets to specify a *CommandText* value in this way. To allow the client dataset to use its *CommandText* property, you must add *poAllowCommandText* to the *Options* property of the provider on the application server. Otherwise, the value of *CommandText* is ignored.

The client dataset sends its *CommandText* string to the provider at two times:

- When the client dataset first opens. After it has retrieved the first data packet from the application server, the client dataset does not send *CommandText* when fetching subsequent data packets.
- When the client dataset sends an *Execute* command to the application server.

To send an SQL command or to change a table or stored procedure name at any other time, you must explicitly use the *IAppServer* interface that is available as the *AppServer* property.

Requesting data from an application server

Topic groups See also

The following table lists the properties and methods of *TClientDataSet* that determine how data is fetched from an application server in a multi-tiered application:

Property or method	Purpose
FetchOnDemand property	Determines whether or not a client dataset automatically fetches data as needed, or relies on the application to call the client dataset's *GetNextPacket*, *FetchBlobs*, and *FetchDetails* functions to retrieve additional data.
PacketRecords property	Specifies the type or number of records to return in each data packet.
GetNextPacket method	Fetches the next data packet from the application server.
FetchBlobs method	Fetches any BLOB fields for the current record when the application server does not include BLOB data automatically.
FetchDetails method	Fetches nested detail datasets for the current record when the application server does not include these in data packets automatically.

By default, a client dataset retrieves all records from the application server. You can control how data is retrieved using *PacketRecords* and *FetchOnDemand*.

PacketRecords specifies either how many records to fetch at a time, or the type of records to return. By default, *PacketRecords* is set to *-1*, which means that all available records are fetched at once, either when the application is first opened, or the application explicitly calls *GetNextPacket*. When *PacketRecords* is *-1*, then after it first fetches data, a client dataset never needs to fetch more data because it already has all available records.

To fetch records in small batches, set *PacketRecords* to the number of records to fetch. For example, the following statement sets the size of each data packet to ten records:

```
ClientDataSet1.PacketRecords := 10;
```

This process of fetching records in batches is called "incremental fetching". Client datasets use incremental fetching when *PacketRecords* is greater than zero. By default, the client dataset calls *GetNextPacket* to fetch data as needed. Newly fetched packets are appended to the end of the data already in the client dataset.

GetNextPacket returns the number of records it fetches. If the return value is the same as *PacketRecords*, the end of available records was not encountered. If the return value is greater than *0* but less than *PacketRecords*, the last record was reached during the fetch operation. If *GetNextPacket* returns *0*, then there are no more records to fetch.

Warning: Incremental fetching only works if the remote data module preserves state information. That is, you must not be using MTS, and the remote data module must be configured so that each client application has its own data module instance. See Supporting state information in remote data modules for information on how to use incremental fetching in stateless remote data modules.

You can also use *PacketRecords* to fetch metadata information about a database from the application server. To retrieve metadata information, set *PacketRecords* to *0*.

Automatic fetching of records is controlled by the *FetchOnDemand* property. When *FetchOnDemand* is *True* (the default), automatic fetching is enabled. To prevent automatic fetching of records as needed, set *FetchOnDemand* to *False*. When *FetchOnDemand* is *false*, the application must explicitly call *GetNextPacket* to fetch records.

Applications that need to represent extremely large read-only datasets can turn off *FetchOnDemand* to ensure that the client datasets do not try to load more data than can fit into memory. Between fetches, the client dataset frees its cache using the *EmptyDataSet* method. This approach, however, does not work well when the client must post updates to the application server.

Handling constraints

Topic groups See also

Client datasets support two types of constraints. These are

- constraints that are sent from the application server in data packets.
- custom constraints provided by the client application.

Handling constraints from the server

Topic groups See also

By default, server constraints and default expressions are passed to client datasets by the application server, where they can be imposed on user data editing. When constraints are in effect, user edits to data in a client application that would violate server constraints are enforced on the client side, and are never passed to the application server for eventual rejection by the database server. This means that fewer updates generate error conditions during the updating process.

While importing of server constraints and expressions is an extremely valuable feature that enables a developer to preserve data integrity across platforms and applications, there may be times when an application needs to disable constraints on a temporary basis. For example, if a server constraint is based on the current maximum value in a field, but the client dataset fetches multiple packets of records, the current maximum value in a field on the client may differ from the maximum value on the database server, and constraints may be invoked differently. In another case, if a client application applies a filter to records when constraints are enabled, the filter may interfere in unintended ways with constraint conditions. In each of these cases, an application may disable constraint-checking.

To disable constraints temporarily, call a client dataset's *DisableConstraints* method. Each time *DisableConstraints* is called, a reference count is incremented. While the reference count is greater than zero, constraints are not enforced on the client dataset.

To reenable constraints for the client dataset, call the dataset's *EnableConstraints* method. Each call to *EnableConstraints* decrements the reference count. When the reference count is zero, constraints are enabled again.

Tip: Always call *DisableConstraints* and *EnableConstraints* in paired blocks to ensure that constraints are enabled when you intend them to be.

Note: *DisableConstraints* and *EnableConstraints* control whether the client dataset applies constraints to its data. However, they have no effect on whether the application server includes constraint information in data packets. You can prevent the server from sending constraints in the first place using the provider's *Constraints* property. For more information on handling constraints from the server side, see Handling server constraints. For more information on working with the constraints once they have been imported, see Using server constraints.

Adding custom constraints

Topic groups See also

You can use the properties of the client dataset's field components to constrain the available values beyond those constraints that are supplied in data packets from the server. Each field component has two properties that you can use to specify constraints:

- The *DefaultExpression* property defines a default value that is assigned to the field if the user does not enter a value. Note that if the application server also assigns a default expression for the field, the client dataset's version takes precedence because it is assigned before the update is returned to the application server.
- The *CustomConstraint* property lets you assign a constraint condition that must be met before a field value can be posted. Custom constraints defined this way are applied in addition to any constraints imported from the server. For more information about working with custom constraints on field components, see Creating a custom constraint.

In addition, you can create record-level constraints using the client dataset's *Constraints* property. *Constraints* is a collection of *TCheckConstraint* objects, where each object represents a separate condition. Use the of *CustomConstraint* property to add your own constraints that are checked when you post records.

Updating records

Topic groups See also

When a client application is connected to an application server, client datasets work with a local copy of data passed to them by the application server. The user sees and edits these copies in the client application's data-aware controls. If server constraints are enabled on the client dataset, a user's edits are constrained accordingly. User changes are temporarily stored by the client dataset in an internally maintained change log. The contents of the change log are stored as a data packet in the *Delta* property. To make the changes in *Delta* permanent, the client dataset must apply them to the database.

When a client applies updates to the server, the following steps occur:

1. The client application calls the *ApplyUpdates* method of a client dataset object. This method passes the contents of the client dataset's *Delta* property to the application server. *Delta* is a data packet that contains a client dataset's updated, inserted, and deleted records.
2. The application server's provider component applies the updates to the database, caching any problem records that it can't resolve at the server level. See Responding to client update requests for details on how the server applies updates.
3. The application server's provider component returns all unresolved records to the client in a *Result* data packet. The *Result* data packet contains all records that were not updated. It also contains error information, such as error messages and error codes.
4. The client application attempts to reconcile update errors returned in the *Result* data packet on a record-by-record basis.

Applying updates

Topic groups See also

Changes made to the client dataset's local copy of data are not sent to the application server until the client application calls the *ApplyUpdates* method for the dataset. *ApplyUpdates* takes the changes in the change log, and sends them as a data packet (called *Delta*) to the application server.

ApplyUpdates takes a single parameter, *MaxErrors*, which indicates the maximum number of errors that the application server should tolerate before aborting the update process. If *MaxErrors* is *0*, then as soon as an update error occurs on the application server, the entire update process is terminated. No changes are written to the database, and the client dataset's change log remains intact. If *MaxErrors* is *-1*, any number of errors is tolerated, and the change log contains all records that could not be successfully applied. If *MaxErrors* is a positive value, and more errors occur than are permitted by *MaxErrors*, all updates are aborted. If fewer errors occur than specified by *MaxErrors*, all records successfully applied are automatically cleared from the client dataset's change log.

ApplyUpdates returns the number of actual errors encountered, which is always less than or equal to *MaxErrors* plus one. This value is set to indicate the number of records it could not write to the database. The application server also returns those records to the client dataset in the dataset.

The client dataset is responsible for reconciling records that generate errors. *ApplyUpdates* calls the *Reconcile* method to write updates to the database. *Reconcile* is an error-handling routine that indirectly calls the *ApplyUpdates* function of a provider component on the application server. The provider component's *ApplyUpdates* function writes the updates to the database and attempts to correct any errors it encounters. Records that it cannot apply because of error conditions are sent back to the client dataset's *Reconcile* method. *Reconcile* then attempts to correct any remaining errors by calling the *OnReconcileError* event handler. You must code the *OnReconcileError* event handler to correct errors. For more information about creating and using *OnReconcileError*, see Reconciling update errors.

Finally, *Reconcile* removes successfully applied changes from the change log and updates *Data* to reflect the newly updated records. When *Reconcile* completes, *ApplyUpdates* reports the number of errors that occurred.

Note: If you are using MTS transactions or sharing an application server instance with other clients, you may want to communicate with the provider on the application server about persistent state information before or after you apply updates. The client dataset receives a *BeforeApplyUpdates* event before the updates are sent which lets you send persistent state information to the server. After the updates are applied (but before the reconcile process), the client dataset receives an *AfterApplyUpdates* event where you can respond to any persistent state information returned by the application server.

Reconciling update errors

Topic groups See also

The provider on the application server returns error records and error information to the client dataset in a result data packet. If the application server returns an error count greater than zero, then for each record in the result data packet, the client dataset's *OnReconcileError* event occurs.

You should always code the *OnReconcileError* event handler, even if only to discard the records returned by the application server. The *OnReconcileError* event handler is passed four parameters:

- *DataSet:* The client dataset to which updates are applied. You can use client dataset methods to obtain information about problem records and to make changes to the record in order to correct any problems. In particular, you will want to use the *CurValue*, *OldValue*, and *NewValue* properties of the fields in the current record to determine the cause of the update problem. However, you must not call any client dataset methods that change the current record in an *OnReconcileError* event handler.
- E: An *EReconcileError* object that represents the problem that occurred. You can use this exception to extract an error message or to determine the cause of the update error.
- *UpdateKind:* The type of update that generated the error. *UpdateKind* can be *ukModify* (the problem occurred updating an existing record that was modified), *ukInsert* (the problem occurred inserting a new record), or *ukDelete* (the problem occurred deleting an existing record).
- *Action:* A **var** parameter that lets you indicate what action to take when the *OnReconcileError* handler exits. On entry into the handler, *Action* is set to the action taken by the resolution process on the server. In your event handler, you set this parameter to
- Skip this record, leaving it in the change log. (raSkip)
- Stop the entire reconcile operation. (raAbort)
- Merge the modification that failed into the corresponding record from the server. (raMerge) This only works if the server did not change any of the fields modified by the user.
- Replace the current update in the change log with the value of the record in the event handler (which has presumably been corrected). (raCorrect)
- Back out the changes for this record on the client dataset, reverting to the originally provided values. (raCancel)
- Update the current record value to match the record on the server. (raRefresh)

The following code shows an *OnReconcileError* event handler that uses the reconcile error dialog from the RecError unit which ships in the object repository directory. (To use this dialog, add RecError to your uses clause.)

```
procedure TForm1.ClientDataSetReconcileError(DataSet: TClientDataSet; E:
EReconcileError; UpdateKind: TUpdateKind, var Action TReconcileAction);
begin
  Action := HandleReconcileError(DataSet, UpdateKind, E);
end;
```

Refreshing records

Topic groups See also

Client applications work with an in-memory snapshot of the data on the application server. As time elapses, other users may modify that data, so that the data in the client application becomes a less and less accurate picture of the underlying data.

Like any other dataset, client datasets have a *Refresh* method that updates its records to match the current values on the server. However, calling *Refresh* only works if there are no edits in the change log. Calling *Refresh* when there are unapplied edits results in an exception.

Client applications can also update the data while leaving the change log intact. To do this, call the client dataset's *RefreshRecord* method. Unlike the *Refresh* method, *RefreshRecord* updates only the current record in the client dataset. *RefreshRecord* changes the record value originally obtained from the application server but leaves any changes in the change log.

Warning: It may not always be appropriate to call *RefreshRecord*. If the user's edits conflict with changes to the underlying dataset made by other users, calling *RefreshRecord* will mask this conflict. When the client application applies its updates, no reconcile error will occur and the application can't resolve the conflict.

In order to avoid masking update errors, client applications may want to check that there are no pending updates before calling *RefreshRecord*. For example, the following code raises an exception if an attempt is made to refresh a modified record:

```
if ClientDataSet1.UpdateStatus <> usUnModified then
  raise Exception.Create('You must apply updates before refreshing the current
record.');
ClientDataSet1.RefreshRecord;
```

Communicating with providers using custom events

Topic groups See also

Client datasets provide many opportunities for customizing the communication between the client application and the application server. Before and after every *IAppServer* method call that is directed at the client dataset's provider, the client dataset receives special events that are designed to allow the client dataset to communicate arbitrary information with its provider. These events are matched with similar events on the provider. Thus for example, when the client dataset calls its *ApplyUpdates* method, the following events occur:

1. The client dataset receives a *BeforeApplyUpdates* event, where it specifies arbitrary custom information in an OleVariant called *OwnerData*.
2. The provider receives a *BeforeApplyUpdates* event, where it can respond to the *OwnerData* from the client dataset and update the value of *OwnerData* to new information.
3. The provider goes through its normal process of assembling a data packet (including all the accompanying events).
4. The provider receives an *AfterApplyUpdates* event, where it can respond to the current value of *OwnerData* and update it to a value for the client.
5. The client dataset receives an *AfterApplyUpdates* event, where it can respond to the returned value of *OwnerData*.

Every other *IAppServer* method call is accompanied by a similar set of *BeforeXXX* and *AfterXXX* events that allow you to customize the communication between client and server.

In addition, the client dataset has a special method, *DataRequest*, whose only purpose is to allow application-specific communication with the provider. When the client dataset calls *DataRequest*, it passes an OleVariant as a parameter that can contain any information you want. This, in turn, generates an is the *OnDataRequest* event on the provider, where you can respond in any application-defined way and return a value to the client dataset.

Using a client dataset with a data provider

Topic groups See also

Client datasets can function independently of a provider, such as in flat-file data base applications and “briefcase model” applications. When there is no provider, however, the client application cannot get table definitions and data from the server, and there is no server to which it can apply updates. Instead, the client dataset must independently

- Define and create tables
- Load saved data
- Merge edits into its data
- Save data

Creating a new dataset

Topic groups See also

There are three ways to define and create client datasets that do not get their data from a provider component:

- You can copy an existing dataset (at design or runtime).
- You can define and create a new client dataset by creating persistent fields for the dataset and then choosing Create Dataset from its context menu. See Creating a new dataset using persistent fields for details.
- You can define and create a new client dataset based on field definitions and index definitions. See Creating a dataset using field and index definitions for details.

Loading data from a file or stream

Topic groups See also

To load data from a file, call a client dataset's *LoadFromFile* method. *LoadFromFile* takes one parameter, a string that specifies the file from which to read data. The file name can be a fully qualified path name, if appropriate. If you always load the client dataset's data from the same file, you can use the *FileName* property instead. If *FileName* names an existing file, the data is automatically loaded when the client dataset is opened.

To load data from a stream, call the client dataset's *LoadFromStream* method. *LoadFromStream* takes one parameter, a stream object that supplies the data.

The data loaded by *LoadFromFile* (*LoadFromStream)* must have previously been saved in a client dataset's data format by this or another client dataset using the *SaveToFile (SaveToStream)* method. For more information about saving data to a file or stream, see Saving data to a file or stream.

When you call *LoadFromFile* or*LoadFromStream*, all data in the file is read into the *Data* property. Any edits that were in the change log when the data was saved are read into the *Delta* property.

Merging changes into Data

Topic groups See also

When you edit the data in a client dataset, the changes you make are recorded in the change log, but the changes do not affect the original version of the data.

To make your changes permanent, call *MergeChangeLog*. *MergeChangeLog* overwrites records in *Data* with any changed field values in the change log.

After *MergeChangeLog* executes, *Data* contains a mix of existing data and any changes that were in the change log. This mix becomes the new *Data* baseline against which further changes can be made. *MergeChangeLog* clears the change log of all records and resets the *ChangeCount* property to *0*.

Warning: Do not call *MergeChangeLog* for client applications that are connected to an application server. In this case, call *ApplyUpdates* to write changes to the database. For more information, see Applying updates.

Note: It is also possible to merge changes into the data of a separate client dataset if that dataset originally provided the data in the *Data* property. To do this, you must use a dataset provider and resolver. For an example of how to do this, see Assigning data directly.

Saving data to a file or stream

Topic groups See also

If you use a client dataset in a single-tiered application, then when you edit data and merge it, the changes you make exist only in memory. To make a permanent record of your changes, you must write them to disk. You can save the data to disk using the *SaveToFile* method.

SaveToFile takes one parameter, a string that specifies the file into which to write data. The file name can be a fully qualified path name, if appropriate. If the file already exists, its current contents are completely overwritten.

If you always save the data to the same file, you can use the *FileName*property instead. If *FileName* is set, the data is automatically saved to the named file when the client dataset is closed.

You can also save data to a stream, using the *SaveToStream* method. *SaveToStream* takes one parameter, a stream object that receives the data.

Note: If you save a client dataset while there are still edits in the change log, these are not merged with the data. When you reload the data, using the *LoadFromFile* or *LoadFromStream* method, the change log will still contain the unmerged edits. This is important for applications that support the briefcase model, where those changes will eventually have to be applied to a provider component on the application server.

Note: *SaveToFile* does not preserve any indexes you added to the client dataset.

Working with cached updates

Topic groups See also

Cached updates enable you to retrieve data from a database, cache and edit it locally, and then apply the cached updates to the database as a unit. When cached updates are enabled, updates to a dataset (such as posting changes or deleting records) are stored in an internal cache instead of being written directly to the dataset's underlying table. When changes are complete, your application calls a method that writes the cached changes to the database and clears the cache.

This section describes when and how to use cached updates. It also describes the TUpdateSQL component that can be used in conjunction with cached updates to update virtually any dataset, particularly datasets that are not normally updatable.

The following topics are discussed in this section:

- Deciding when to use cached updates
- Using cached updates
- Using update objects to update a dataset
- Updating a read-only result set
- Controlling the update process
- Handling cached update errors

Deciding when to use cached updates

Topic groups See also

Cached updates are primarily intended to reduce data access contention on remote database servers by

- Minimizing transaction times.
- Minimizing network traffic.

While cached updates can minimize transaction times and drastically reduce network traffic, they may not be appropriate for all database client applications that work with remote servers. There are three areas of consideration when deciding to use cached updates:

- **Cached data is local to your application, and is not under transaction control.** In a busy client/server environment this has two implications for your application:
- Other applications can access and change the actual data on the server while your users edit their local copies of the data.
- Other applications cannot see any data changes made by your application until it applies all its changes.
- **In master/detail relationships managing the order of applying cached updates can be tricky.** This is particularly true when there are nested master/detail relationships where one detail table is the master table for yet another detail table and so on.
- **Applying cached updates to read-only query-based datasets requires use of update objects.**

The data access components provide cached update methods and transaction control methods you can use in your application code to handle these situations, but you must take care that you cover all possible scenarios your application is likely to encounter in your working environment.

Using cached updates

Topic groups See also

This section provides a basic overview of how cached updates work in an application. If you have not used cached updates before, this process description serves as a guideline for implementing cached updates in your applications.

To use cached updates, the following order of processes must occur in an application:

1. **Enable cached updates.** Enabling cached updates causes a read-only transaction that fetches as much data from the server as is necessary for display purposes and then terminates. Local copies of the data are stored in memory for display and editing.
2. **Display and edit the local copies of records,** permit insertion of new records, and support deletions of existing records. Both the original copy of each record and any edits to it are stored in memory.
3. **Fetch additional records as necessary.** As a user scrolls through records, additional records are fetched as needed. Each fetch occurs within the context of another short duration, read-only transaction. (An application can optionally fetch all records at once instead of fetching many small batches of records.)
4. **Continue to display and edit local copies of records** until all desired changes are complete.
5. **Apply the locally cached records to the database** or cancel the updates. For each record written to the database, an *OnUpdateRecord* event is triggered. If an error occurs when writing an individual record to the database, an *OnUpdateError* event is triggered which enables the application to correct the error, if possible, and continue updating. When updates are complete, all successfully applied updates are cleared from the local cache.

   If instead of applying updates, an application cancels updates, the locally cached copy of the records and all changes to them are freed without writing the changes to the database.

For specific descriptions of these operations, see

- Enabling and disabling cached updates.
- Fetching records.
- Applying cached updates.
- Canceling pending cached updates.
- Undeleting cached records.
- Specifying visible records in the cache.
- Checking update status.

Enabling and disabling cached updates

Topic groups See also

Cached updates are enabled and disabled through the CachedUpdates properties of *TTable*, *TQuery*, and *TStoredProc*. *CachedUpdates* is *False* by default, meaning that cached updates are not enabled for a dataset.

Note: Client datasets always cache updates. They have no *CachedUpdates* property because you cannot disable cached updates on a client dataset.

To use cached updates, set *CachedUpdates* to *True*, either at design time (through the Object Inspector), or at runtime. When you set *CachedUpdates* to *True*, the dataset's *OnUpdateRecord* event is triggered if you provide it. For more information about the *OnUpdateRecord* event, see Creating an OnUpdateRecord event handler.

For example, the following code enables cached updates for a dataset at runtime:

```
CustomersTable.CachedUpdates := True;
```

When you enable cached updates, a copy of all records necessary for display and editing purposes is cached in local memory. Users view and edit this local copy of data. Changes, insertions, and deletions are also cached in memory. They accumulate in memory until the current cache of local changes is applied to the database. If changed records are successfully applied to the database, the record of those changes are freed in the cache.

Note: Applying cached updates does not disable further cached updates; it only writes the current set of changes to the database and clears them from memory.

To disable cached updates for a dataset, set *CachedUpdates* to *False*. If you disable cached updates when there are pending changes that you have not yet applied, those changes are discarded without notification. Your application can test the *UpdatesPending* property for this condition before disabling cached updates. For example, the following code prompts for confirmation before disabling cached updates for a dataset:

```
if (CustomersTable.UpdatesPending)
  if (Application.MessageBox("Discard pending updates?",
                             "Unposted changes",
                             MB_YES + MB_NO) = IDYES) then
    CustomersTable.CachedUpdates = False;
```

Fetching records

Topic groups See also

By default, when you enable cached updates, BDE datasets automatically handle fetching of data from the database when necessary. Datasets fetch enough records for display. During the course of processing, many such record fetches may occur. If your application has specific needs, it can fetch all records at one time. You can fetch all records by calling the dataset's *FetchAll* method. *FetchAll* creates an in-memory, local copy of all records from the dataset. If a dataset contains many records or records with large BLOB fields, you may not want to use *FetchAll*.

Client datasets use the *PacketRecords* property to indicate the number of records that should be fetched at any time. If you set the *FetchOnDemand* property to *True*, the client dataset automatically handles fetching of data when necessary. Otherwise, you can use the *GetNextPacket* method to fetch records from the data server. For more information about fetching records using a client dataset, see Requesting data from an application server

Applying cached updates

Topic groups See also

When a dataset is in cached update mode, changes to data are not actually written to the database until your application explicitly calls methods that apply those changes. Normally an application applies updates in response to user input, such as through a button or menu item.

Important: To apply updates to a set of records retrieved by an SQL query that does not return a live result set, you must use a *TUpdateSQL* object to specify how to perform the updates. For updates to joins (queries involving two or more tables), you must provide one *TUpdateSQL* object for each table involved, and you must use the *OnUpdateRecord* event handler to invoke these objects to perform the updates. For more information, see Updating a read-only result set. For more information about creating and using an *OnUpdateRecord* event handler, see Creating an OnUpdateRecord event handler.

Applying updates is a two-phase process that should occur in the context of a database component's transaction control to enable your application to recover gracefully from errors. For more information about transaction handling with database components, see Understanding database and session component interactions.

When applying updates under database transaction control, the following events take place:

1. A database transaction starts.
2. Cached updates are written to the database (phase 1). If you provide it, an *OnUpdateRecord* event is triggered once for each record written to the database. If an error occurs when a record is applied to the database, the *OnUpdateError* event is triggered if you provide one.

   If the database write is unsuccessful:

   - Database changes are rolled back, ending the database transaction.
   - Cached updates are not committed, leaving them intact in the internal cache buffer.

   If the database write is successful:

   - Database changes are committed, ending the database transaction.
   - Cached updates are committed, clearing the internal cache buffer (phase 2).

The two-phased approach to applying cached updates allows for effective error recovery, especially when updating multiple datasets (for example, the datasets associated with a master/detail form). For more information about handling update errors that occur when applying cached updates, see Handling cached update errors.

There are actually two ways to apply updates. To apply updates for a specified set of datasets associated with a database component, call the database component's *ApplyUpdates* method. To apply updates for a single dataset, call the dataset's *ApplyUpdates* and *Commit* methods.

Applying cached updates with a database component method.

Applying cached updates with dataset component methods.

Applying updates for master/detail tables.

Applying cached updates with a database component method

Topic groups See also

Ordinarily, applications cache updates at the dataset level. However, there are times when it is important to apply the updates to multiple interrelated datasets in the context of a single transaction. For example, when working with master/detail forms, you will likely want to commit changes to master and detail tables together.

To apply cached updates to one or more datasets in the context of a database connection, call the database component's *ApplyUpdates* method. The following code applies updates to the *CustomersQuery* dataset in response to a button click event:

```
procedure TForm1.ApplyButtonClick(Sender: TObject);
begin
  // for local databases such as Paradox, dBASE, and FoxPro
  // set TransIsolation to DirtyRead
  if not(Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
    Database1.TransIsolation := tiDirtyRead;
  Database1.ApplyUpdates([CustomersQuery]);
end;
```

The above sequence starts a transaction, and writes cached updates to the database. If successful, it also commits the transaction, and then commits the cached updates. If unsuccessful, this method rolls back the transaction, and does not change the status of the cached updates. In this latter case, your application should handle cached update errors through a dataset's *OnUpdateError* event. For more information about handling update errors, see Handling cached update errors.

The main advantage to calling a database component's *ApplyUpdates* method is that you can update any number of dataset components that are associated with the database. The parameter for the *ApplyUpdates* method for a database is an array of *TDBDataSet*. For example, the following code applies updates for two queries used in a master/detail form:

```
if not(Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead) then
  Database1.TransIsolation := tiDirtyRead;
Database1.ApplyUpdates([CustomerQuery, OrdersQuery]);
```

For more information about updating master/detail tables, see Applying updates for master/detail tables.

Applying cached updates with dataset component methods

Topic groups See also

You can apply updates for individual datasets directly using the dataset's *ApplyUpdates* and *CommitUpdates* methods. Each of these methods encapsulate one phase of the update process:

1 *ApplyUpdates* writes cached changes to a database (phase 1).
2 *CommitUpdates* clears the internal cache when the database write is successful (phase 2).

Applying updates at the dataset level gives you control over the order in which updates are applied to individual datasets. Order of update application is especially critical for handling master/detail relationships. To ensure the correct ordering of updates for master/detail tables, you should always apply updates at the dataset level. For more information see Applying updates for master/detail tables.

The following code illustrates how you apply updates within a transaction for the *CustomerQuery* dataset previously used to illustrate updates through a database method:

```
procedure TForm1.ApplyButtonClick(Sender: TObject)
begin
  Database1.StartTransaction;
  try
    if not(Database1.IsSQLBased) and not (Database1.TransIsolation = tiDirtyRead)
then
      Database1.TransIsolation := tiDirtyRead;
    CustomerQuery.ApplyUpdates;               { try to write the updates to the
database }
    Database1.Commit;                                  { on success, commit the
changes }
  except
    Database1.Rollback;                                  { on failure, undo any
changes }
    raise;                { raise the exception again to prevent a call to
CommitUpdates }
  end;
  CustomerQuery.CommitUpdates;                    { on success, clear the internal
cache }
end;
```

If an exception is raised during the *ApplyUpdates* call, the database transaction is rolled back. Rolling back the transaction ensures that the underlying database table is not changed. The *raise* statement inside the try...except block re-*raises* the exception, thereby preventing the call to *CommitUpdates*. Because *CommitUpdates* is not called, the internal cache of updates is not cleared so that you can handle error conditions and possibly retry the update.

Applying updates for master/detail tables

Topic groups See also

When you apply updates for master/detail tables, the order in which you list datasets to update is significant. Generally you should always update master tables before detail tables, except when handling deleted records. In complex master/detail relationships where the detail table for one relationship is the master table for another detail table, the same rule applies.

You can update master/detail tables at the database or dataset component levels. For purposes of control (and of creating explicitly self-documented code), you should apply updates at the dataset level. The following example illustrates how you should code cached updates to two tables, *Master* and *Detail*, involved in a master/detail relationship:

```
Database1.StartTransaction;
try
  Master.ApplyUpdates;
  Detail.ApplyUpdates;
  Database1.Commit;
except
  Database1.Rollback;
  raise;
end;
Master.CommitUpdates;
Detail.CommitUpdates;
```

If an error occurs during the application of updates, this code also leaves both the cache and the underlying data in the database tables in the same state they were in before the calls to *ApplyUpdates*.

If an exception is *raised* during the call to *Master.ApplyUpdates*, it is handled like the single dataset case previously described. Suppose, however, that the call to *Master.ApplyUpdates* succeeds, and the subsequent call to *Detail.ApplyUpdates* fails. In this case, the changes are already applied to the master table. Because all data is updated inside a database transaction, however, even the changes to the master table are rolled back when *Database1.Rollback* is called in the except block. Furthermore, *UpdatesMaster.CommitUpdates* is not called because the exception which is re-*raised* causes that code to be skipped, so the cache is also left in the state it was before the attempt to update.

To appreciate the value of the two-phase update process, assume for a moment that *ApplyUpdates* is a single-phase process which updates the data *and the cache*. If this were the case, and if there were an error while applying the updates to the *Detail* table, then there would be no way to restore both the data and the cache to their original states. Even though the call to *Database1.Rollback* would restore the database, there would be no way to restore the cache.

Canceling pending cached updates

Topic groups See also

Pending cached updates are updated records that are posted to the cache but not yet applied to the database. There are three ways to cancel pending cached updates:

- To cancel all pending updates and disable further cached updates, set the *CachedUpdates* property to *False*. For more information, see Canceling pending updates and disabling further cached updates
- To discard all pending updates without disabling further cached updates, call the *CancelUpdates* method. For more information, see Canceling pending cached updates
- To cancel updates made to the current record call *RevertRecord*. For more information, see Canceling updates to the current record

Canceling pending updates and disabling further cached updates

Topic groups See also

To cancel further caching of updates and delete all pending cached updates without applying them, set the *CachedUpdates* property to *False*. When *CachedUpdates* is set to *False*, the *CancelUpdates* method is automatically invoked.

From the update cache, deleted records are undeleted, modified records revert to original values, and newly inserted record simply disappear.

Note: This option is not available for client datasets.

Canceling pending cached updates

Topic groups See also

CancelUpdates clears the cache of all pending updates, and restores the dataset to the state it was in when the table was opened, cached updates were last enabled, or updates were last successfully applied. For example, the following statement cancels updates for the *CustomersTable*:

```
CustomersTable.CancelUpdates;
```

From the update cache, deleted records are undeleted, modified records revert to original values, and newly inserted records simply disappear.

Note: Calling *CancelUpdates* does not disable cached updating. It only cancels currently pending updates. To disable further cached updates, set the *CachedUpdates* property to *False*.

Canceling updates to the current record

Topic groups See also

RevertRecord restores the current record in the dataset to the state it was in when the table was opened, cached updates were last enabled, or updates were last successfully applied. It is most frequently used in an *OnUpdateError* event handler to correct error situations. For example,

```
CustomersTable.RevertRecord;
```

Undoing cached changes to one record does not affect any other records. If only one record is in the cache of updates and the change is undone using *RevertRecord*, the *UpdatesPending* property for the dataset component is automatically changed from *True* to *False*.

If the record is not modified, this call has no effect. For more information about creating an *OnUpdateError* handler, see Creating an OnUpdateRecord event handler.

Undeleting cached records

Topic groups See also

To undelete a cached record requires some coding because once the deleted record is posted to the cache, it is no longer the current record and no longer even appears in the dataset. In some instances, however, you may want to undelete such records. The process involves using the *UpdateRecordTypes* property to make the deleted records “visible,” and then calling *RevertRecord*. Here is a code example that undeletes all deleted records in a table:

```
procedure TForm1.UndeleteAll(DataSet: TBDEDataSet)
begin
  DataSet.UpdateRecordTypes := [rtDeleted];                  { show only deleted
records }
  try
    DataSet.First;                             { go to the first previously deleted
record }
    while not (DataSet.Eof)
      DataSet.RevertRecord;                      { undelete until we reach the last
record }
  except
             { restore updates types to recognize only modified, inserted, and
unchanged }
    DataSet.UpdateRecordTypes := [rtModified, rtInserted, rtUnmodified];
    raise;
  end;
  DataSet.UpdateRecordTypes := [rtModified, rtInserted, rtUnmodified];
end;
```

For more information on *UpdateRecordTypes*, see Specifying visible records in the cache.

Specifying visible records in the cache

Topic groups See also

The *UpdateRecordTypes* property controls what type of records are visible in the cache when cached updates are enabled. *UpdateRecordTypes* works on cached records in much the same way as filters work on tables. *UpdateRecordTypes* is a set, so it can contain any combination of the following values:

Value	Meaning
rtModified	Modified records
rtInserted	Inserted records
rtDeleted	Deleted records
rtUnmodified	Unmodified records

The default value for *UpdateRecordTypes* includes only *rtModified, rtInserted,* and *rtUnmodified*, with deleted records (*rtDeleted*) not displayed.

The *UpdateRecordTypes* property is primarily useful in an *OnUpdateError* event handler for accessing deleted records so they can be undeleted through a call to *RevertRecord*. This property is also useful if you wanted to provide a way in your application for users to view only a subset of cached records, for example, all newly inserted (*rtInserted*) records.

For example, you could have a set of four radio buttons (*RadioButton1* through *RadioButton4*) with the captions All, Modified, Inserted, and Deleted. With all four radio buttons assigned to the same *OnClick* event handler, you could conditionally display all records (except deleted, the default), only modified records, only newly inserted records, or only deleted records by appropriately setting the *UpdateRecordTypes* property.

```
procedure TForm1.UpdateFilterRadioButtonsClick(Sender: TObject);
begin
  if RadioButton1.Checked then
    CustomerQuery.UpdateRecordTypes := [rtUnmodified, rtModified, rtInserted]
  else if RadioButton2.Checked then
    CustomerQuery.UpdateRecordTypes := [rtModified]
  else if RadioButton3.Checked then
    CustomerQuery.UpdateRecordTypes := [rtInserted]
  else
    CustomerQuery.UpdateRecordTypes := [rtDeleted];
end;
```

For more information about creating an *OnUpdateError* handler, see Creating an OnUpdateRecord event handler.

Checking update status

Topic groups See also

When cached updates are enabled for your application, you can keep track of each pending update record in the cache by examining the *UpdateStatus* property for the record. Checking update status is most frequently used in *OnUpdateRecord* and *OnUpdateError* event handlers. For more information about creating and using an *OnUpdateRecord* event, see Creating an OnUpdateRecord event handler. For more information about creating and using an *OnUpdateError* event, see Handling cached update errors.

As you iterate through a set of pending changes, *UpdateStatus* changes to reflect the update status of the current record. *UpdateStatus* returns one of the following values for the current record:

Value	Meaning
usUnmodified	Record is unchanged
usModified	Record is changed
usInserted	Record is a new record
usDeleted	Record is deleted

When a dataset is first opened all records will have an update status of *usUnmodified*. As records are inserted, deleted, and so on, the status values change. Here is an example of *UpdateStatus* property used in a handler for a dataset's *OnScroll* event. The event handler displays the update status of each record in a status bar.

```
procedure TForm1.CustomerQueryAfterScroll(DataSet: TDataSet);
begin
  with CustomerQuery do begin
    case UpdateStatus of
      usUnmodified: StatusBar1.Panels[0].Text := 'Unmodified';
      usModified: StatusBar1.Panels[0].Text := 'Modified';
      usInserted: StatusBar1.Panels[0].Text := 'Inserted';
      usDeleted: StatusBar1.Panels[0].Text := 'Deleted';
      else StatusBar1.Panels[0].Text := 'Undetermined status';
    end;
  end;
end;
```

Note: If a record's *UpdateStatus* is *usModified*, you can examine the *OldValue* property for each field in the dataset to determine its previous value. *OldValue* is meaningless for records with *UpdateStatus* values other than *usModified*. For more information about examining and using *OldValue*, see Accessing a field's OldValue, NewValue, and CurValue properties.

Using update objects to update a dataset

Topic groups See also

TUpdateSQL is an update component that uses SQL statements to update a dataset. You must provide one *TUpdateSQL* component for each underlying table accessed by the original query that you want to update.

Note: If you use more than one update component to perform an update operation, you must create an *OnUpdateRecord* event to execute each update component.

An update component actually encapsulates three *TQuery* components. Each of these query components perform a single update task. One query component provides an SQL UPDATE statement for modifying existing records; a second query component provides an INSERT statement to add new records to a table; and a third component provides a DELETE statement to remove records from a table.

When you place an update component in a data module, you do not see the query components it encapsulates. They are created by the update component at runtime based on three update properties for which you supply SQL statements:

- *ModifySQL* specifies the UPDATE statement.
- *InsertSQL* specifies the INSERT statement.
- *DeleteSQL* specifies the DELETE statement.

At runtime, when the update component is used to apply updates, it:

1. Selects an SQL statement to execute based on the *UpdateKind* parameter automatically generated on a record update event. *UpdateKind* specifies whether the current record is modified, inserted, or deleted.
2. Provides parameter values to the SQL statement.
3. Prepares and executes the SQL statement to perform the specified update.

The following topics are discussed in this section:

- Specifying the UpdateObject property for a dataset
- Creating SQL statements for update components
- Executing update statements
- Using dataset components to update a dataset

Specifying the UpdateObject property for a dataset

Topic groups See also

One or more update objects can be associated with a dataset to be updated. Associate update objects with the update dataset either by setting the dataset component's *UpdateObject* property to the update object or by setting the update object's *DataSet* property to the update dataset. Which method is used depends on whether only one base table in the update dataset is to be updated or multiple tables.

You must use one of these two means of associating update datasets with update objects. Without proper association, the dynamic filling of parameters in the update object's SQL statements cannot occur. Use one association method or the other, but never both.

How an update object is associated with a dataset also determines how the update object is executed. An update object might be executed automatically, without explicit intervention by the application, or it might need to be explicitly executed. If the association is made using the dataset component's *UpdateObject* property, the update object will automatically be executed. If the association is made with the update object's *DataSet* property, you must programmatically execute the update object.

The sections that follow explain the process of associating update objects with update dataset components in greater detail, along with suggestions about when each method should be used and effects on update execution.

- Using a single update object
- Using multiple update objects

Using a single update object

Topic groups See also

When only one of the base tables referenced in the update dataset needs to be updated, associate an update object with the dataset by setting the dataset component's *UpdateObject* property to the name of the update object.

```
Query1.UpdateObject := UpdateSQL1;
```

The update SQL statements in the update object are automatically executed when the update dataset's *ApplyUpdates* method is called. The update object is invoked for each record that requires updating. Do not call the update object's *ExecSQL* method in a handler for the *OnUpdateRecord* event as this will result in a second attempt to apply each record's update.

If you supply a handler for the dataset's *OnUpdateRecord* event, the minimum action that you need to take in that handler is setting the event handler's *UpdateAction* parameter to *uaApplied*. You may optionally perform data validation, data modification, or other operations like setting parameter values.

Using multiple update objects

Topic groups See also

When more than one base table referenced in the update dataset needs to be updated, you need to use multiple update objects: one for each base table updated. Because the dataset component's *UpdateObject* only allows one update object to be associated with the dataset, you must associate each update object with the dataset by setting its *DataSet* property to the name of the dataset. The *DataSet* property for update objects is not available at design time in the Object Inspector. You can only set this property at runtime.

```
UpdateSQL1.DataSet := Query1;
```

The update SQL statements in the update object are not automatically executed when the update dataset's *ApplyUpdates* method is called. To update records, you must supply a handler for the dataset component's *OnUpdateRecord* event and call the update object's *ExecSQL* or *Apply* method. This invokes the update object for each record that requires updating.

In the handler for the dataset's *OnUpdateRecord* event, the minimum actions that you need to take in that handler are:

- Calling the update object's *SetParams* method (if you later call *ExecSQL*).
- Executing the update object for the current record with *ExecSQL* or *Apply*.
- Setting the event handler's *UpdateAction* parameter to *uaApplied*.

You may optionally perform data validation, data modification, or other operations that depend on each record's update.

Note: It is also possible to have one update object associated with the dataset using the dataset component's *UpdateObject* property, and the second and subsequent update objects associated using their *DataSet* properties. The first update object is executed automatically on calling the dataset component's *ApplyUpdates* method. The rest need to be manually executed.

Creating SQL statements for update components

Topic groups See also

To update a record in an associated dataset, an update object uses one of three SQL statements. The three SQL statements delete, insert, and modify records cached for update. The statements are contained in the update object's string list properties *DeleteSQL*, *InsertSQL*, and *ModifySQL*. As each update object is used to update a single table, the object's update statements each reference the same base table.

As the update for each record is applied, one of the three SQL statements is executed against the base table updated. Which SQL statement is executed depends on the *UpdateKind* parameter automatically generated for each record's update.

Creating the SQL statements for update objects can be done at design time or at runtime. The sections that follow describe the process of creating update SQL statements in greater detail.

- Creating SQL statements at design time
- Understanding parameter substitution in update SQL statements
- Composing update SQL statements
- Using an update component's Query property
- Using the DeleteSQL, InsertSQL, and ModifySQL properties

Creating SQL statements at design time

Topic groups See also

To create the SQL statements for an update component,

1. Select the *TUpdateSQL* component.
2. Select the name of the update component from the drop-down list for the dataset component's *UpdateObject* property in the Object Inspector. This step ensures that the Update SQL editor you invoke in the next step can determine suitable default values to use for SQL generation options.
3. Right-click the update component and select UpdateSQL Editor from the context menu to invoke the Update SQL editor. The editor creates SQL statements for the update component's *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties based on the underlying data set and on the values you supply to it.

The Update SQL editor has two pages. The Options page is visible when you first invoke the editor. Use the Table Name combo box to select the table to update. When you specify a table name, the Key Fields and Update Fields list boxes are populated with available columns.

The Update Fields list box indicates which columns should be updated. When you first specify a table, all columns in the Update Fields list box are selected for inclusion. You can multi-select fields as desired.

The Key Fields list box is used to specify the columns to use as keys during the update. For Paradox, dBASE, and FoxPro the columns you specify here must correspond to an existing index, but this is not a requirement for remote SQL databases. Instead of setting Key Fields you can click the Primary Keys button to choose key fields for the update based on the table's primary index. Click Dataset Defaults to return the selection lists to the original state: all fields selected as keys and all selected for update.

Check the Quote Field Names check box if your server requires quotation marks around field names.

After you specify a table, select key columns, and select update columns, click Generate SQL to generate the preliminary SQL statements to associate with the update component's *ModifySQL*, *InsertSQL*, and *DeleteSQL* properties. In most cases you may want or need to fine tune the automatically generated SQL statements.

To view and modify the generated SQL statements, select the SQL page. If you have generated SQL statements, then when you select this page, the statement for the *ModifySQL* property is already displayed in the SQL Text memo box. You can edit the statement in the box as desired.

Important: Keep in mind that generated SQL statements are starting points for creating update statements. You may need to modify these statements to make them execute correctly. For example, when working with data that contains NULL values, you need to modify the WHERE clause to read

```
WHERE field IS NULL
```

rather then using the generated field variable. Test each of the statements directly yourself before accepting them.

Use the Statement Type radio buttons to switch among generated SQL statements and edit them as desired.

To accept the statements and associate them with the update component's SQL properties, click OK.

For information on using parameter substitution in generated SQL statements, see Understanding parameter substitution in update SQL statements.

Understanding parameter substitution in update SQL statements

Topic groups See also

Update SQL statements use a special form of parameter substitution that enables you to substitute old or new field values in record updates. When the Update SQL editor generates its statements, it determines which field values to use. When you write the update SQL, you specify the field values to use.

When the parameter name matches a column name in the table, the new value in the field in the cached update for the record is automatically used as the value for the parameter. When the parameter name matches a column name prefixed by the string “OLD_”, then the old value for the field will be used. For example, in the update SQL statement below, the parameter :LastName is automatically filled with the new field value in the cached update for the inserted record.

```
INSERT INTO Names
(LastName, FirstName, Address, City, State, Zip)
VALUES (:LastName, :FirstName, :Address, :City, :State, :Zip)
```

New field values are typically used in the *InsertSQL* and *ModifySQL* statements. In an update for a modified record, the new field value from the update cache is used by the UPDATE statement to replace the old field value in the base table updated.

In the case of a deleted record, there are no new values, so the *DeleteSQL* property uses the “:OLD_FieldName” syntax. Old field values are also normally used in the WHERE clause of the SQL statement for a modified or deletion update to determine which record to update or delete.

In the WHERE clause of an UPDATE or DELETE update SQL statement, supply at least the minimal number of parameters to uniquely identify the record in the base table that is updated with the cached data. For instance, in a list of customers, using just a customer’s last name may not be sufficient to uniquely identify the correct record in the base table; there may be a number of records with “Smith” as the last name. But by using parameters for last name, first name, and phone number could be a distinctive enough combination. Even better would be a unique field value like a customer number.

For more information about old and new value parameter substitution, see Accessing a field’s OldValue, NewValue, and CurValue properties.

Composing update SQL statements

Topic groups See also

The *TUpdateSQL* component has three properties for updating SQL statements: *DeleteSQL*, *InsertSQL*, and *ModifySQL*. As the names of the properties imply, these SQL statements delete, insert, and modify records in the base table.

The *DeleteSQL* property should contain only an SQL statement with the DELETE command. The base table to be updated must be named in the FROM clause. So that the SQL statement only deletes the record in the base table that corresponds to the record deleted in the update cache, use a WHERE clause. In the WHERE clause, use a parameter for one or more fields to uniquely identify the record in the base table that corresponds to the cached update record. If the parameters are named the same as the field and prefixed with “OLD_”, the parameters are automatically given the values from the corresponding field from the cached update record. If the parameter are named in any other manner, you must supply the parameter values.

```
DELETE FROM Inventory I
WHERE (I.ItemNo = :OLD_ItemNo)
```

Some tables types might not be able to find the record in the base table when fields used to identify the record contain NULL values. In these cases, the delete update fails for those records. To accommodate this, add a condition for those fields that might contain NULLs using the IS NULL predicate (in addition to a condition for a non-NULL value). For example, when a FirstName field may contain a NULL value:

```
DELETE FROM Names
WHERE (LastName = :OLD_LastName) AND
  ((FirstName = :OLD_FirstName) OR (FirstName IS NULL))
```

The *InsertSQL* statement should contain only an SQL statement with the INSERT command. The base table to be updated must be named in the INTO clause. In the VALUES clause, supply a comma-separated list of parameters. If the parameters are named the same as the field, the parameters are automatically given the value from the cached update record. If the parameter are named in any other manner, you must supply the parameter values. The list of parameters supplies the values for fields in the newly inserted record. There must be as many value parameters as there are fields listed in the statement.

```
INSERT INTO Inventory
(ItemNo, Amount)
VALUES (:ItemNo, 0)
```

The *ModifySQL* statement should contain only an SQL statement with the UPDATE command. The base table to be updated must be named in the FROM clause. Include one or more value assignments in the SET clause. If values in the SET clause assignments are parameters named the same as fields, the parameters are automatically given values from the fields of the same name in the updated record in the cache. You can assign additional field values using other parameters, as long as the parameters are not named the same as any fields and you manually supply the values. As with the *DeleteSQL* statement, supply a WHERE clause to uniquely identify the record in the base table to be updated using parameters named the same as the fields and prefixed with “OLD_”. In the update statement below, the parameter :ItemNo is automatically given a value and :Price is not.

```
UPDATE Inventory I
SET I.ItemNo = :ItemNo, Amount = :Price
WHERE (I.ItemNo = :OLD_ItemNo)
```

Considering the above update SQL, take an example case where the application end-user modifies an existing record. The original value for the ItemNo field is 999. In a grid connected to the cached dataset, the end-user changes the ItemNo field value to 123 and Amount to 20. When the ApplyUpdates method is invoked, this SQL statement affects all records in the base table where the ItemNo field is 999, using the old field value in the parameter :OLD_ItemNo. In those records, it changes the ItemNo field value to 123 (using the parameter :ItemNo, the value coming from the grid) and Amount to 20.

Accessing an update component's Query property

Topic groups See also

Use the *Query* property of an update component to access one of the update SQL properties *DeleteSQL*, *InsertSQL*, or *ModifySQL*, such as to set or alter the SQL statement. Use *UpdateKind* constant values as an index into the array of query components. The *Query* property is only accessible at runtime.

The statement below uses the *UpdateKind* constant *ukDelete* with the *Query* property to access the *DeleteSQL* property.

```
with UpdateSQL1.Query[ukDelete] do begin
  Clear;
  Add('DELETE FROM Inventory I');
  Add('WHERE (I.ItemNo = :OLD_ItemNo)');
end;
```

Normally, the properties indexed by the *Query* property are set at design time using the Update SQL editor. You might, however, need to access these values at runtime if you are generating a unique update SQL statement for each record and not using parameter binding. The following example generates a unique *Query* property value for each row updated:

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  with UpdateSQL1 do begin
    case UpdateKind of
      ukModified:
        begin
          Query[UpdateKind].Text := Format('update emptab set Salary = %d where
EmpNo = %d',
            [EmpAuditSalary.NewValue, EmpAuditEmpNo.OldValue]);
          ExecSQL(UpdateKind);
        end;
      ukInserted:
        {...}
      ukDeleted:
        {...}
    end;
  end;
  UpdateAction := uaApplied;
end;
```

Note: *Query* returns a value of type *TDataSetUpdateObject*. To treat this return value as a TUpdateSQL component, to use properties and methods specific to TUpdateSQL, typecast the *UpdateObject* property. For example:

```
with (DataSet.UpdateObject as TUpdateSQL).Query[UpdateKind] do begin
  { perform operations on the statement in DeleteSQL }
end;
```

For an example of using this property, see Calling the SetParams method.

Using the DeleteSQL, InsertSQL, and ModifySQL properties

Topic groups See also

Use the *DeleteSQL*, *InsertSQL*, and *ModifySQL* properties to set the update SQL statements for each. These properties are all string list containers. Use the methods of string lists to enter SQL statement lines as items in these properties. Use an integer value as an index to reference a specific line within the property. The *DeleteSQL*, *InsertSQL*, and *ModifySQL* properties are accessible both at design time and at runtime.

```
with UpdateSQL1.DeleteSQL do begin
  Clear;
  Add('DELETE FROM Inventory I');
  Add('WHERE (I.ItemNo = :OLD_ItemNo)');
end;
```

Below, the third line of an SQL statement is altered using an index of 2 with the *ModifySQL* property.

```
UpdateSQL1.ModifySQL[2] := 'WHERE ItemNo = :ItemNo';
```

Executing update statements

Topic groups See also

There are a number of methods involved in executing the update SQL for an individual record update. These method calls are typically used within a handler for the *OnUpdateRecord* event of the update object to execute the update SQL to apply the current cached update record. The various methods are applicable under different circumstances. The sections that follow discuss each of the methods in detail.

- Calling the Apply method
- Calling the SetParams method
- Calling the ExecSQL method

Calling the Apply method

Topic groups See also

The *Apply* method for an update component manually applies updates for the current record. There are two steps involved in this process:

1. Values for the record are bound to the parameters in the appropriate update SQL statement.
2. The SQL statement is executed.

Call the *Apply* method to apply the update for the current record in the update cache. Only use *Apply* when the update object is not associated with the dataset using the dataset component's *UpdateObject* property, in which case the update object is not automatically executed. *Apply* automatically calls the *SetParams* method to bind old and new field values to specially named parameters in the update SQL statement. Do not call *SetParams* yourself when using *Apply*. The *Apply* method is most often called from within a handler for the dataset's *OnUpdateRecord* event.

If you use the dataset component's UpdateObject property to associate dataset and update object, this method is called automatically. Do not call *Apply* in a handler for the dataset component's *OnUpdateRecord* event as this will result in a second attempt to apply the current record's update.

In a handler for the *OnUpdateRecord* event, the *UpdateKind* parameter is used to determine which update SQL statement to use. If invoked by the associated dataset, the *UpdateKind* is set automatically. If you invoke the method in an *OnUpdateRecord* event, pass an *UpdateKind* constant as the parameter of *Apply*.

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  UpdateSQL1.Apply(UpdateKind);
  UpdateAction := uaApplied;
end;
```

If an exception is raised during the execution of the update program, execution continues in the *OnUpdateError* event, if it is defined.

Note: The operations performed by *Apply* are analogous to the *SetParams* and *ExecSQL* methods described in the following sections.

Calling the SetParams method

Topic groups See also Example

The *SetParams* method for an update component uses special parameter substitution rules to substitute old and new field values into the update SQL statement. Ordinarily, *SetParams* is called automatically by the update component's *Apply* method. If you call *Apply* directly in an *OnUpdateRecord* event, do not call *SetParams* yourself. If you execute an update object using its *ExecSQL* method, call *SetParams* to bind values to the update statement's parameters.

SetParams sets the parameters of the SQL statement indicated by the *UpdateKind* parameter. Only those parameters that use a special naming convention automatically have a value assigned. If the parameter has the same name as a field or the same name as a field prefixed with "OLD_" the parameter is automatically a value. Parameters named in other ways must be manually assigned values. For more information see the section Understanding parameter substitution in update SQL statements.

Example: Using SetParams

The following example illustrates one such use of *SetParams*:

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  with DataSet.UpdateObject as TUpdateSQL do begin
    SetParams(UpdateKind);
    if UpdateKind = ukModified then
      Query[UpdateKind].ParamByName('DateChanged').Value := Now;
    ExecSQL(UpdateKind);
  end;
  UpdateAction := uaApplied;
end;
```

Note: This example assumes that the *ModifySQL* property for the update component is as follows:

```
UPDATE EmpAudit
SET EmpNo = :EmpNo, Salary = :Salary, Changed = :DateChanged
WHERE EmpNo = :OLD_EmpNo
```

In this example, the call to *SetParams* supplies values to the *EmpNo* and *Salary* parameters. The *DateChanged* parameter is not set because the name does not match the name of a field in the dataset, so the next line of code sets this value explicitly.

Executing an update statement

Topic groups See also

The *ExecSQL* method for an update component manually applies updates for the current record. There are two steps involved in this process:

1. Values for the record are bound to the parameters in the appropriate update SQL statement.
2. The SQL statement is executed.

Call the *ExecSQL* method to apply the update for the current record in the update cache. Only use *ExecSQL* when the update object is not associated with the dataset using the dataset component's *UpdateObject* property, in which case the update object is not automatically executed. *ExecSQL* does not automatically call the *SetParams* method to bind update SQL statement parameter values; call *SetParams* yourself before invoking *ExecSQL*. The *ExecSQL* method is most often called from within a handler for the dataset's *OnUpdateRecord* event.

If you use the dataset component's *UpdateObject* property to associate dataset and update object, this method is called automatically. Do not call *ExecSQL* in a handler for the dataset component's *OnUpdateRecord* event as this will result in a second attempt to apply the current record's update.

In a handler for the *OnUpdateRecord* event, the *UpdateKind* parameter is used to determine which update SQL statement to use. If invoked by the associated dataset, the *UpdateKind* is set automatically. If you invoke the method in an *OnUpdateRecord* event, pass an *UpdateKind* constant as the parameter of *ExecSQL*.

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  with (DataSet.UpdateObject as TUpdateSQL) do begin
    SetParams(UpdateKind);
    ExecSQL(UpdateKind);
  end;
  UpdateAction := uaApplied;
end;
```

If an exception is raised during the execution of the update program, execution continues in the *OnUpdateError* event, if it is defined.

Note: The operations performed by *ExecSQL* and *SetParams* are analogous to the *Apply* method described previously.

Using dataset components to update a dataset

Topic groups See also Example

Applying cached updates usually involves use of one or more update objects. The update SQL statements for these objects apply the data changes to the base table. Using update components is the easiest way to update a dataset, but it is not a requirement. You can alternately use dataset components like *TTable* and *TQuery* to apply the cached updates.

In a handler for the dataset component's OnUpdateRecord event, use the properties and methods of another dataset component to apply the cached updates for each record.

Example: Using a TTable to perform the updates

For example, the following code uses a table component to perform updates:

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  if UpdateKind = ukInsert then
    UpdateTable.AppendRecord([DataSet.Fields[0].NewValue,
DataSet.Fields[1].NewValue])
  else
    if UpdateTable.Locate('KeyField', VarToStr(DataSet.Fields[1].OldValue), []) then
      case UpdateKind of
        ukModify:
          begin
            Edit;
            UpdateTable.Fields[1].AsString := VarToStr(DataSet.Fields[1].NewValue);
            Post;
          end;
        ukInsert:
          begin
            Insert;
            UpdateTable.Fields[1].AsString := VarToStr(DataSet.Fields[1].NewValue);
            Post;
          end;
        ukModify: DeleteRecord;
      end;
        UpdateAction := uaApplied;
end;
```

Updating a read-only result set

Topic groups See also

Although the Borland Database Engine (BDE) attempts to provide an updatable, or “live” query result when the *RequestLive* property for a *TQuery* component is *True*, there are some situations where it cannot do so. (For more information see the section Deciding when to use cached updates.)

In these situations, you can manually update a dataset as follows:

1. Add a *TUpdateSQL* component to the data module in your application.
2. Set the dataset component’s *UpdateObject* property to the name of the *TUpdateSQL* component in the data module.
3. Enter the SQL update statement for the result set to the update component’s *ModifySQL*, *InsertSQL*, or *DeleteSQL* properties, or use the Update SQL editor.
4. Close the dataset.
5. Set the dataset component’s *CachedUpdates* property to *True*.
6. Reopen the dataset.

Note: In many circumstances, you may also want to write an *OnUpdateRecord* event handler for the dataset.

Controlling the update process

Topic groups See also

When a dataset component's *ApplyUpdates* method is called, an attempt is made to apply the updates for all records in the update cache to the corresponding records in the base table. As the update for each changed, deleted, or newly inserted record is about to be applied, the dataset component's *OnUpdateRecord* event fires.

Providing a handler for the *OnUpdateRecord* event allows you to perform actions just before the current record's update is actually applied. Such actions can include special data validation, updating other tables, or executing multiple update objects. A handler for the *OnUpdateRecord* event affords you greater control over the update process.

The sections that follow describe when you might need to provide a handler for the *OnUpdateRecord* event and how to create a handler for this event.

- Determining if you need to control the updating process
- Creating an OnUpdateRecord event handler

Determining if you need to control the updating process

Topic groups See also

Some of the time when you use cached updates, all you need to do is call *ApplyUpdates* to apply cached changes to the base tables in the database (for example, when you have exclusive access to a local Paradox or dBASE table through a *TTable* component). In most other cases, however, you either might want to or must provide additional processing to ensure that updates can be properly applied. Use a handler for the updated dataset component's *OnUpdateRecord* event to provide this additional processing.

For example, you might want to use the *OnUpdateRecord* event to provide validation routines that adjust data before it is applied to the table, or you might want to use the *OnUpdateRecord* event to provide additional processing for records in master and detail tables before writing them to the base tables.

In many cases you must provide additional processing. For example, if you access multiple tables using a joined query, then you must provide one *TUpdateSQL* object for each table in the query, and you must use the *OnUpdateRecord* event to make sure each update object is executed to write changes to the tables.

For more information on creating and using a *TUpdateSQL* object, see Using update objects to update a dataset. For more information on creating and using an *OnUpdateRecord* event handler, see Creating an OnUpdateRecord event handler.

Creating an OnUpdateRecord event handler

Topic groups See also Example

The *OnUpdateRecord* event handles cases where a single update component cannot be used to perform the required updates, or when your application needs more control over special parameter substitution. The *OnUpdateRecord* event fires once for the attempt to apply the changes for each modified record in the update cache.

To create an *OnUpdateRecord* event handler for a dataset:

1. Select the dataset component.
2. Choose the Events page in the Object Inspector.
3. Double-click the *OnUpdateRecord* property value to invoke the code editor.

Here is the skeleton code for an *OnUpdateRecord* event handler:

```
procedure TForm1.DataSetUpdateRecord(DataSet: TDataSet;
 UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  { perform updates here... }
end;
```

The *DataSet* parameter specifies the cached dataset with updates.

The *UpdateKind* parameter indicates the type of update to perform. Values for *UpdateKind* are *ukModify*, *ukInsert*, and *ukDelete*. When using an update component, you need to pass this parameter to its execution and parameter binding methods. For example using *ukModify* with the *Apply* method executes the update object's *ModifySQL* statement. You may also need to inspect this parameter if your handler performs any special processing based on the kind of update to perform.

The *UpdateAction* parameter indicates if you applied an update or not. Values for *UpdateAction* are *uaFail* (the default), *uaAbort*, *uaSkip*, *uaRetry*, *uaApplied*. Unless you encounter a problem during updating, your event handler should set this parameter to *uaApplied* before exiting. If you decide not to update a particular record, set the value to *uaSkip* to preserve unapplied changes in the cache.

If you do not change the value for *UpdateAction*, the entire update operation for the dataset is aborted. For more information about *UpdateAction*, see Specifying the action to take.

In addition to these parameters, you will typically want to make use of the *OldValue* and *NewValue* properties for the field component associated with the current record. For more information about *OldValue* and *NewValue,* see Accessing a field's OldValue, NewValue, and CurValue properties.

Important: The *OnUpdateRecord* event, like the *OnUpdateError* and *OnCalcFields* event handlers, should never call any methods that change which record in a dataset is the current record.

Example: An OnUpdateRecord event handler

Here is an *OnUpdateRecord* event handler that executes two update components using their *Apply* methods. The *UpdateKind* parameter is passed to the *Apply* method to determine which update SQL statement in each update object to execute.

```
procedure TForm1.EmpAuditUpdateRecord(DataSet: TDataSet;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  EmployeeUpdateSQL.Apply(UpdateKind);
  JobUpdateSQL.Apply(UpdateKind);
  UpdateAction := uaApplied;
end;
```

In this example the *DataSet* parameter is not used. This is because the update components are not associated with the dataset component using its *UpdateObject* property.

Handling cached update errors

Topic groups See also

Because there is a delay between the time a record is first cached and the time cached updates are applied, there is a possibility that another application may change the record in a database before your application applies its updates. Even if there is no conflict between user updates, errors can occur when a record's update is applied. The Borland Database Engine (BDE) specifically checks for user update conflicts and other conditions when attempting to apply updates, and reports an error.

A dataset component's *OnUpdateError* event enables you to catch and respond to errors. You should create a handler for this event if you use cached updates. If you do not, and an error occurs, the entire update operation fails.

Caution: Do not call any dataset methods that change the current record (such as *Next* and *Prior*) in an *OnUpdateError* event handler. Doing so causes the event handler to enter an endless loop.

Here is the skeleton code for an *OnUpdateError* event handler:

```
procedure TForm1.DataSetUpdateError(DataSet: TDataSet; E: EDatabaseError;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  { ... perform update error handling here ... }
end;
```

The following sections describe specific aspects of error handling using an *OnUpdateError* handler, and how the event's parameters are used.

- Referencing the dataset to which to apply updates
- Indicating the type of update that generated an error
- Specifying the action to take
- Working with error message text
- Accessing a field's OldValue, NewValue, and CurValue properties

Referencing the dataset to which to apply updates

Topic groups See also

DataSet references the dataset to which updates are applied. To process new and old record values during error handling you must supply this reference.

Indicating the type of update that generated an error

Topic groups See also Example

The OnUpdateRecord event receives the parameter *UpdateKind*, which of type *TUpdateKind*. It describes the type of update that generated the error. Unless your error handler takes special actions based on the type of update being carried out, your code probably will not make use of this parameter.

The following table lists possible values for *UpdateKind*:

Value	Meaning
ukModify	Editing an existing record caused an error.
ukInsert	Inserting a new record caused an error.
ukDelete	Deleting an existing record caused an error.

Example: Using the UpdateKind parameter

The example below shows the decision construct to perform different operations based on the value of the *UpdateKind* parameter.

```
procedure TForm1.DataSetUpdateError(DataSet: TDataSet; E: EDatabaseError;
  UpdateKind: TUpdateKind; var UpdateAction: TUpdateAction);
begin
  case UpdateKind of
    ukModify:
      begin
        { handle error due to applying record modification update }
      end;
    ukInsert:
      begin
        { handle error due to applying record insertion update }
      end;
    ukDelete:
      begin
        { handle error due to applying record deletion update }
      end;
  end;
end;
```

Specifying the action to take

Topic groups See also

UpdateAction is a parameter of type *TUpdateAction*. When your update error handler is first called, the value for this parameter is always set to *uaFail*. Based on the error condition for the record that caused the error and what you do to correct it, you typically set *UpdateAction* to a different value before exiting the handler. *UpdateAction* can be set to one of the following values:

Value	Meaning
uaAbort	Aborts the update operation without displaying an error message.
uaFail	Aborts the update operation, and displays an error message. This is the default value for *UpdateAction* when you enter an update error handler.
uaSkip	Skips updating the row, but leaves the update for the record in the cache.
uaRetry	Repeats the update operation. Correct the error condition before setting *UpdateAction* to this value.
uaApplied	Not used in error handling routines.

If your error handler can correct the error condition that caused the handler to be invoked, set *UpdateAction* to the appropriate action to take on exit. For error conditions you correct, set *UpdateAction* to *uaRetry* to apply the update for the record again.

When set to *uaSkip*, the update for the row that caused the error is skipped, and the update for the record remains in the cache after all other updates are completed.

Both *uaFail* and *uaAbort* cause the entire update operation to end. *uaFail* raises an exception and displays an error message. *uaAbort* raises a silent exception (does not display an error message).

Note: If an error occurs during the application of cached updates, an exception is *raised* and an error message displayed. Unless the *ApplyUpdates* is called from within a try...except construct, an error message to the user displayed from inside your *OnUpdateError* event handler may cause your application to display the same error message twice. To prevent error message duplication, set *UpdateAction* to *uaAbort* to turn off the system-generated error message display.

The *uaApplied* value should only be used inside an *OnUpdateRecord* event. Do not set this value in an update error handler. For more information about update record events, see Creating an OnUpdateRecord event handler.

Working with error message text

Topic groups See also

The E parameter is usually of type *EDBEngineError*. From this exception type, you can extract an error message that you can display to users in your error handler. For example, the following code could be used to display the error message in the caption of a dialog box:

```
ErrorLabel.Caption := E.Message;
```

This parameter is also useful for determining the actual cause of the update error. You can extract specific error codes from *EDBEngineError*, and take appropriate action based on it. For example, the following code checks to see if the update error is related to a key violation, and if it is, it sets the *UpdateAction* parameter to *uaSkip*:

```
{ Add 'Bde' to your uses clause for this example }
if (E is EDBEngineError) then
  with EDBEngineError(E) do begin
    if Errors[ErrorCount - 1].ErrorCode = DBIERR_KEYVIOL then
      UpdateAction := uaSkip                    { key violation, just skip this
record }
    else
      UpdateAction := uaAbort;               { don't know what's wrong, abort the
update }
  end;
```

Accessing a field's OldValue, NewValue, and CurValue properties

Topic groups See also Example

When cached updates are enabled for records, the original values for fields in each record are stored in a read-only *TField* property called *OldValue*. Changed values are stored in the analogous *TField* property *NewValue*. In client datasets, an additional *TField* property, *CurValue*, stores the field values that currently appear in the dataset. *CurValue* will be the same as *OldValue* unless another user has edited the record. If another user has edited the record, *CurValue* will give the current value of the field which was posted by that user.

These values provide the only way to inspect and change update values in *OnUpdateError* and *OnUpdateRecord* event handlers. For more information about *OnUpdateRecord*, see Creating an OnUpdateRecord event handler.

Example: Using the OldValue, NewValue, and CurValue properties

In some cases, you may be able to use the *OldValue*, *NewValue*, and *CurValue* properties to determine the cause of an error and correct it. For example, the following code handles corrections to a salary field which can only be increased by 25 percent at a time (enforced by a constraint on the server):

```
var
  SalaryDif: Integer;
  OldSalary: Integer;
begin
  OldSalary := EmpTabSalary.OldValue;
  SalaryDif := EmpTabSalary.NewValue - OldSalary;
  if SalaryDif / OldSalary > 0.25 then begin
    { Increase was too large, drop it back to 25% }
    EmpTabSalary.NewValue := OldSalary + OldSalary * 0.25;
    UpdateAction := uaRetry;
  end
  else
    UpdateAction := uaSkip;
end;
```

NewValue is decreased to a 25 percent increase in the case where the cached update would have increased salary by a larger percentage. Then the update operation is retried. To improve the efficiency of this routine, the *OldValue* parameter is stored in a local variable.

Using data controls

Topic groups See also

A *data-aware* control derives display data from a database source outside the application, and can optionally post (or return) data changes to a data source.

Using common data control features describes basic features common to all data control components.

Most data-aware components represent information stored in a dataset. These can be grouped into controls that represent a single field and controls that represent sets of records, such as DBGrids and control grids. In addition, the navigator control, *TDBNavigator*, is a visual tool that lets your users navigate and manipulate records.

More complex data-aware controls for decision support are discussed in Using decision support components.

Using common data control features

Topic groups See also

The following tasks are common to most data controls:

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

You place data controls from the Data Controls page of the Component palette onto the forms in your database application. Data controls generally enable you to display and edit fields of data associated with the current record in a dataset. The following table summarizes the data controls that appear on the Data Controls page of the Component palette.

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

Data controls are data-aware at design time. When you set a control's *DataSource* property 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 a dataset at design time to verify that your application displays data correctly without having to compile and run the application. For more information about the Fields editor, see "Creating persistent fields."

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

Associating a data control with a dataset

Topic groups See also

Data controls connect to datasets by using a data source. A data source component acts as a conduit between the control and a dataset containing data.

To associate a data control with a dataset,

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

Editing and updating data

Topic groups See also

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

The following topics describe how to allow users to edit data using data controls:

- Enabling editing in controls on user entry
- Editing data in a control

Enabling editing in controls on user entry

Topic groups See also

A dataset must be in *dsEdit* state to permit editing to its data. The AutoEdit property of the data source to which a control is attached determines if the underlying dataset enters *dsEdit* mode when data in a control is modified in response to keyboard or mouse events. When *AutoEdit* is *True* (the default), *dsEdit* mode is set as soon as editing commences. If *AutoEdit* is *False*, you must provide a TDBNavigator control with an *Edit* button (or some other method) to permit users to set *dsEdit* state at runtime. For more information about *TDBNavigator*, see "Navigating and manipulating records."

Editing data in a control

Topic groups See also

The *ReadOnly* property of a data control determines if a user can edit the data displayed by the control. If *False* (the default), users can edit data. To prevent users from editing data in a control, set *ReadOnly* to *True*.

Properties of the data source and dataset underlying a control also determine if the user can successfully edit data with a control and can post changes to the dataset.

The Enabled property of a data source determines if controls attached to a data source are able to display fields values from the dataset, and therefore also determine if a user can edit and post values. If *Enabled* is *True* (the default), controls can display field values.

The ReadOnly property of the dataset determines if user edits can be posted to the dataset. If *False* (the default), changes are posted to the dataset. If *True*, the dataset is read-only.

Note: An additional read-only, runtime property CanModify determines if a dataset can be modified. *CanModify* is set to *True* if a database permits write access. If *CanModify* is *False*, a dataset is read-only. Query components that perform inserts and updates are, by definition, able to write to an underlying database, provided that your application and user have sufficient write privileges to the database itself.

The following table summarizes the factors that determine if a user can edit data in a control and post changes to the database.

Component	**Property**				
Data control	ReadOnly	false	false	false	true
Data Source	Enabled	true	true	false	—
Dataset	ReadOnly	false	false	—	—
Dataset	CanModify	true	false	—	—
(database)	write access	Read/Write	Read-only	—	—
Can write to database?		Yes	No	No	No

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

In TDBGrid, modifications are 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.

Disabling and enabling data display

Topic groups See also

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

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

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

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

Refreshing data

Topic groups See also

The Refresh method for a dataset flushes local buffers and refetches data for an open dataset. You can use this method to update the display in data-aware controls if you think that the underlying data has changed because other applications have simultaneous access to the data used in your application.

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

Enabling mouse, keyboard, and timer events

Topic groups See also

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

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

Using data sources

Topic groups See also

A *TDataSource* component is a nonvisual database component that acts as a conduit between a dataset and data-aware components on a form that enable the display, navigation, and editing of the data underlying the dataset. Each data-aware control needs to be associated with a data source component to have data to display and manipulate. Similarly, all datasets must be associated with a data source component in order for their data to be displayed and manipulated in data-aware controls on a form.

Note: You can also use data source components to link datasets in master-detail relationships.

Place a data source component in a data module or form just as you place other nonvisual database components. You should place at least one data source component for each dataset component in a data module or form.

The following topics are discussed in this section:

- Using TDataSource properties
- Using TDataSource events

Using TDataSource properties

Topic groups See also

TDataSource has only a few published properties. The following sections discuss these key properties and how to set them at design time and runtime.

- Setting the DataSet property
- Setting the Name property
- Setting the Enabled property
- Setting the AutoEdit property

Setting the DataSet property

Topic groups See also

The *DataSet* property specifies the dataset from which a data source component gets its data. At design time, you can select a dataset from the drop-down list in the Object Inspector. At runtime, you can switch the dataset for a data source component as needed. For example, the following code swaps the dataset for the *CustSource* data source component between *Customers* and *Orders*:

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

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

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

Setting the Name property

Topic groups See also

Name enables you to specify a meaningful name for a data source component that distinguishes it from all other data sources in your application. The name you supply for a data source component is displayed below the component's icon in a data module.

Generally, you should provide a name for a data source component that indicates the dataset with which it is associated. For example, suppose you have a dataset called *Customers*, and that you link a data source component to it by setting the data source component's *DataSet* property to "Customers." To make the connection between the dataset and data source obvious in a data module, you should set the *Name* property for the data source component to something like "CustomersSource."

Setting the Enabled property

Topic groups See also

The *Enabled* property determines if a data source component is connected to its dataset. When *Enabled* is *True*, the data source is connected to a dataset.

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

Setting the AutoEdit property

Topic groups See also

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), the dataset automatically enters 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 about dataset states, see "Determining and setting dataset states".

Using TDataSource events

Topic groups See also

TDataSource has three event handlers associated with it:

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

Using the OnDataChange event

Topic groups See also

An *OnDataChange* event occurs whenever the cursor moves to a new record. When an application calls *Next, Previous, Insert*, or any method that leads to a change in the cursor position, an *OnDataChange* is triggered.

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

Using the OnUpdateData event

Topic groups See also

An *OnUpdateData* event occurs whenever the data in the current record is about to be updated. For instance, an *OnUpdateData* event occurs 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.

Using the OnStateChange event

Topic groups See also

An *OnStateChange* event occurs whenever the state for a data source's dataset changes. A dataset's *State* property records its current state. *OnStateChange* is useful for performing actions as a *TDataSource*'s state changes.

For example, during the course of a normal database session, a dataset's state changes frequently. To track these changes, you could write an *OnStateChange* event handler that displays the current dataset state in a label on a form. The following code illustrates one way you might code such a routine. At runtime, this code displays the current setting of the dataset's *State* property and updates it every time it changes:

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

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

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

Controls that represent a single field

Topic groups See also

Many of the controls on the data controls page of the component palette represent a single field in a database table. Most of these controls are similar in appearance and function to standard Windows controls that you place on forms. For example, the *TDBEdit* control is a data-aware version of the standard *TEdit* control which enables users to see and edit a text string.

Which control you use depends on the type of data (text, formatted text, graphics, boolean information, and so on) contained in the field. The following topics describe these components in more detail:

- Displaying and editing fields in an edit box
- Displaying and editing text in a memo control
- Displaying and editing text in a rich edit memo control
- Displaying and editing graphics fields in an image control
- Displaying and editing data in list and combo boxes
- Displaying and editing data in lookup list and combo boxes
- Handling Boolean field values with check boxes
- Restricting field values with radio controls

Displaying data as labels

Topic groups See also

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

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

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

Displaying and editing fields in an edit box

Topic groups See also

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

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

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

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

Displaying and editing text in a memo control

Topic groups See also

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. You can use *TDBMemo* controls to display memo fields from dBASE and Paradox tables and text data contained in BLOB fields.

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

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

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

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

Displaying and editing text in a rich edit memo control

Topic groups See also

TDBRichEdit is a data-aware component—similar to the standard *TRichEdit* component—that can display formatted text stored in a binary large object (BLOB) field. *TDBMemo*TDBMemo displays formatted, multi-line text, and permits a user to enter formatted multi-line text as well. You can use *TDBRichEdit* controls to display memo fields from dBASE and Paradox tables and text data contained in BLOB fields.

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

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

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

Displaying and editing graphics fields in an image control

Topic groups See also

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.

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

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

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

Displaying and editing data in list and combo boxes

Topic groups See also

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 runtime.

Note: Data-aware list and combo box can be linked only to data sources for table components. They do not work with query components.

The following table describes these controls.

Data control	Description
TDBListBox	Displays a list of items from which a user can update a field in the current record. The list of display items is a property of the control.
TDBComboBox	Combines an edit box with a list box. A user can update a field in the current record by choosing a value from the drop-down list or by entering a value. The list of display items is a property of the control.
TDBLookupListBox	Displays a list of items from which a user can update a column in the current record. The list of display items is looked up in another dataset.
TDBLookupComboBox	Combines an edit box with a list box. A user can update a field in the current record by choosing a value from the drop-down list or by entering a value. The list of display items is looked up in another dataset.

The following topics are discussed in this section:

- Displaying and editing data in a list box
- Displaying and editing data in a combo box

Displaying and editing data in a list box

Topic groups See also

TDBListBox displays a scrollable list of items from which a user can choose to enter in a data field. A data-aware list box displays a default value for a field in the current record and highlights its corresponding entry in the list. If the current row's field value is not in the list, no value is highlighted in the list box. When a user selects a list item, the corresponding field value is changed in the underlying dataset.

Use the String List editor at design time to create the list of items to display in the *Items* property. The *Height* property determines how many items are visible in the list box at one time. The *IntegralHeight* property controls the way the list box is displayed. If *IntegralHeight* is *False* (the default), the bottom of the list box is determined by the *ItemHeight* property, and the bottom item might not be completely displayed. If *IntegralHeight* is *True* the visible bottom item in the list box is fully displayed.

Displaying and editing data in a combo box

Topic groups See also

The TDBComboBox control combines the functionality of a data-aware edit control and a drop-down list. At runtime it can display a drop-down list from which a user can pick from a predefined set of values, and it can permit a user to enter an entirely different value.

The *Items* property of the component specifies the items contained in the drop-down list. At design time, use the String List Editor to populate the *Items* list. At runtime, use the methods of the *Items* property to manipulate its string list.

When a control is linked to a field through its *DataField* property, it displays the value for the field in the current row, regardless of whether it appears in the *Items* list. The *Style* property determines user interaction with the control. By default, *Style* is *csDropDown*, meaning a user can enter values from the keyboard, or choose an item from the drop-down list. The following properties determine how the *Items* list is displayed at runtime:

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

Displaying and editing data in lookup list and combo boxes

Topic groups See also

TDBLookupListBox and TDBLookupComboBox are data-aware controls that derive a list of display items from one of two sources:

- Lookup field defined for a dataset.
- Secondary data source, data field, and key.

In either case, a user is presented with a restricted list of choices from which to set a valid field value. When a user selects a list item, the corresponding field value is changed in the underlying dataset.

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

The following topics are discussed in this section:

- Specifying a list based on a lookup field
- Specifying a list based on a secondary data source
- Setting lookup list and combo box properties
- Searching incrementally for list item values

Specifying a list based on a lookup field

Topic groups See also

To specify list box items using a lookup field, the dataset to which you link the control must already define a lookup field. For more information about defining a lookup field for a dataset, see "Defining a lookup field."

To specify a lookup field for the list box items,

1. Set the *DataSource* property of the list box to the data source for the dataset containing the lookup field to use.
2. Choose the lookup field to use from the drop-down list for the *DataField* property.

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

Specifying a list based on a secondary data source

Topic groups See also

If you have not defined a lookup field for a dataset, you can establish a similar relationship using a secondary data source, a field value to search on in the secondary data source, and a field value to return as a list item.

To specify a secondary data source for list box items,

1. Set the *DataSource* property of the list box to the data source for the control.
2. Choose a field into which to insert looked-up values from the drop-down list for the *DataField*property. The field you choose cannot be a lookup field.
3. Set the *ListSource* property of the list box to the data source for the dataset that contain the field whose values you want to look up.
4. Choose a field to use as a lookup key from the drop-down list for the *KeyField* property. The drop-down list displays fields for the dataset associated with data source you specified in Step 3. The field you choose need not be part of an index, but if it is, lookup performance is even faster.
5. Choose a field whose values to return from the drop-down list for the *ListField* property. The drop-down list displays fields for the dataset associated with the data source you specified in Step 3.

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

Setting lookup list and combo box properties

Topic groups See also

The following table lists important properties for lookup list and combo boxes.

Property	Purpose
DataField	Specifies the field in the master dataset which provides the key value to be looked up in the lookup dataset. This field is modified when a user selects a list box or combo box item. If *DataField* is set to a lookup field, the *KeyField*, *ListField*, and *ListSource* properties are not used.
DataSource	Specifies a data source for the control. If the selection in the control is changed, this dataset is placed in *dsEdit* mode.
KeyField	Specifies the field in the lookup dataset corresponding to *DataField*. The control searches for the *DataField* value in the *KeyField* of the lookup dataset. The lookup dataset should have an index on this field to facilitate lookups.
ListField	Specifies the field of the lookup dataset to display in the control.
ListSource	Specifies a data source for the secondary (lookup) dataset. The sort order of items displayed in the list box or combo box is determined by the index specified by the *IndexName* property of the lookup dataset. That index need not be the same one used by the *KeyField* property.
RowCount	**TDBLookupListBox only**. Specifies the number of lines of text to display in the list box. The height of the list box is adjusted to fit this row count exactly.
DropDownRows	**TDBLookupComboBox only.** Specifies the number of lines of text to display in the drop-down list.

Searching incrementally for list item values

Topic groups See also

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

Handling Boolean field values with check boxes

Topic groups See also

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

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

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

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

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

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

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

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

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

Restricting field values with radio controls

Topic groups See also

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

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

If the current value of a field associated with a radio group matches one of the strings in the *Items* property, that radio button is selected. For example, if three strings, "Red," "Yellow," and "Blue," are listed for *Items*, and the field for the current record contains the value "Blue," then the third button in the group 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, suppose *Items* contains "Red," "Yellow," and "Blue," and *Values* contains "Magenta," "Yellow," and "Cyan." If a user selects the button labeled "Red," "Magenta" is posted to the database.

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

Viewing and editing data with TDBGrid

Topic groups See also

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

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

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

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

The *State*property of a grid's *Columns* property indicates if persistent column objects exist for the grid. *Columns.State* is a runtime only property that is automatically set for a grid. The default state is *csDefault*, meaning that persistent column objects do not exist for the grid. In that case, the display of data in the grid is determined either by persistent field components for the dataset displayed in the grid, or for datasets without persistent field components, by a default set of data display characteristics.

The following topics are discussed in this section:

- Using a grid control in its default state
- Creating a customized grid
- Setting grid options
- Displaying ATD and array fields
- Editing in the grid
- Rearranging column order at design time
- Rearranging column order at runtime
- Controlling grid drawing
- Responding to user actions at runtime

Using a grid control in its default state

Topic groups See also

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

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

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

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

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

Creating a customized grid

Topic groups See also

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

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

The following topics are discussed in this section:

- Understanding persistent columns
- Determining the source of a column property at runtime

Understanding persistent columns

Topic groups See also

When you create persistent column objects for a grid, they are only loosely associated with underlying fields in a grid's dataset. Default property values for persistent columns are dynamically fetched from a default source (such as the grid or associated field) until a value is assigned to the column property. Until you assign a column property a value, its value changes as its default source changes.

For example, the default source for a column title caption is an associated field's *DisplayLabel* property. If you modify the *DisplayLabel* property, the column title reflects that change immediately.

Once you assign a value to a column property, it no longer changes when its default source changes. For example, if you assign a string to the column title's caption, the title caption is independent of the associated field's *DisplayLabel* property. Subsequent changes to the field's *DisplayLabel* property no longer affect the column's title.

Persistent columns exist independently from field components with which they are associated. In fact, persistent columns do not have to be associated with field objects at all. If a persistent column's *FieldName* property is blank, or if the field name does not match the name of any field in the grid's current dataset, the column's *Field* property is NULL and the column is drawn with blank cells. You can use a blank column to display bitmaps or bar charts that graphically depict some aspect of a record's data in an otherwise blank cell, for example. To do so, you must override the cells' default drawing method.

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

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

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

The following topics are discussed in this section:

- Creating persistent columns
- Deleting persistent columns
- Arranging the order of persistent columns
- Defining a lookup list column
- Defining a pick list column
- Putting a button in a column
- Setting column properties at design time
- Restoring default values to a column

Determining the source of a column property at runtime

Topic groups See also

At runtime you can test a column's *AssignedValues* property to determine whether a column property gets its values from an associated field component, or is assigned its own value.

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

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

To add a persistent column call the *Add* method for the column list:

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

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

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

Finally, assigning *csCustomized* to the *Column.State* property for a grid at runtime puts the grid into customized mode. Any existing columns in the grid are destroyed and new persistent columns are built for each field in the grid's dataset.

Creating persistent columns

Topic groups See also

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

To create persistent columns for a grid control,

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

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

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

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

To create persistent columns individually:

1. Choose the Add button in the Columns editor. The new column will be selected in the list box. The new column is given a sequential number and default name (for example, 0 - TColumn).
2. To associate a field with this new column, set the *FieldName* property in the Object Inspector.
3. To set the title for the new column, set the *Caption* option for the *Title* property in the Object Inspector.
4. Close the Columns editor to apply the persistent columns to the grid and close the dialog box.

Deleting persistent columns

Topic groups See also

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

1. Select the field to remove in the Columns list box.
2. Click Delete (you can also use the context menu or *Del* key, to remove a column).

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

Arranging the order of persistent columns

Topic groups See also

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

To change the order of a column,

1. Select the column in the Columns list box.
2. Drag it to a new location in the list box.

You can also change the column order by dragging the column in the actual grid, just as you can at runtime.

Defining a lookup list column

Topic groups See also

To make a column display a drop-down list of values drawn from a separate lookup table, you must define a lookup field object in the dataset. For more information about creating lookup fields, see Defining a lookup field.

Once the lookup field is defined, set the column's *FieldName* to the lookup field name and make sure the column's *ButtonStyle* is set to *cbsAuto*. The grid automatically displays a combo box-like drop-down button when a cell of that column is in edit mode. The drop-down list is populated with lookup values defined by the lookup field.

Defining a pick list column

Topic groups See also

A pick list column looks and operates like a lookup list column, except that the column's field is a normal field and the drop-down list is populated with the list of values in the column's *PickList* property instead of from a lookup table.

To define a pick list column:

1. Select the column in the *Columns* list box.
2. Set *ButtonStyle* to *cbsAuto*.
3. Double-click the Picklist property in the Object Inspector to bring up a string list editor.

In the String List editor, enter the list of values you want to appear in the drop-down list for this column. If the pick list contains data, the column becomes a pick list column.

Note: To restore a column to its normal behavior, delete all the text from the Pick list editor.

Putting a button in a column

Topic groups See also

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

To create an ellipsis button in a column:

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

Setting column properties at design time

Topic groups See also

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

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

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

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

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

Restoring default values to a column

Topic groups See also

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

Displaying ADT and array fields

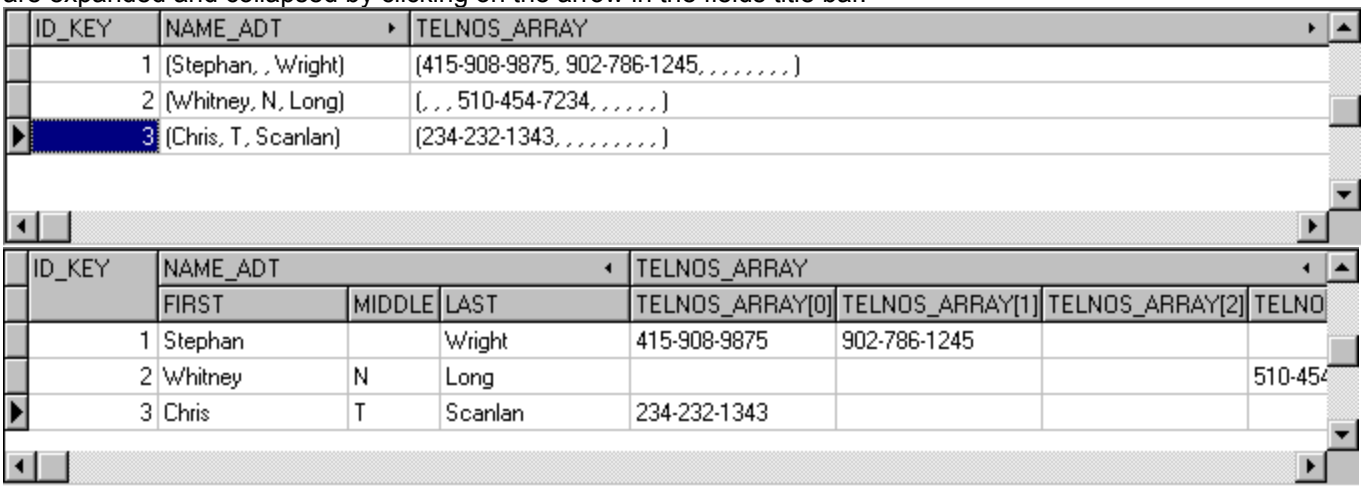
Topic groups See also

Depending on the value of the dataset's *ObjectView* property, a grid displays ADT and array fields either flattened out, or in an object mode, where the field can be expanded and collapsed. When *ObjectView* is *True*, the object fields can be expanded and collapsed. When a field is expanded, each child field appears in its own column with a title bar, which are below the title bar of the ADT or array field itself. When the field is collapsed, only one column appears with an uneditable comma delimited string containing the child fields. A column can be expanded and collapsed by clicking on the arrow in the title bar of the field, and by setting the Expanded property of the column. When the dataset's *ObjectView* property is *False*, each child field appears in a separate column.

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

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

The following figures show the grid with an ADT field and an array field. The first figure shows the fields collapsed. In this state they cannot be edited. The second figure shows the fields expanded. The fields are expanded and collapsed by clicking on the arrow in the fields title bar.



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

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

Setting grid options

Topic groups See also

You can use the grid *Options* property at design time to control basic grid behavior and appearance at runtime. When a grid component is first placed on a form at design time, the *Options* property in the Object Inspector is displayed with a + (plus) sign to indicate that the *Options* property can be expanded to display a series of Boolean properties that you can set individually.

To view and set these properties, 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 runtime.

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

	records. *False*: Permits pending inserts.
dgMultiSelect	*True*: Allows user to select noncontiguous rows in the grid using *Ctrl*+*Shift* or *Shift*+ *arrow* keys. *False*: (Default). Does not allow user to multi-select rows.

Editing in the grid

Topic groups See also

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

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

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

Topic groups See also

In grid controls with persistent columns, and default grids whose datasets contain persistent fields, you can reorder the grid columns at design time by clicking on the title cell of a column and dragging it to its new location in the grid.

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

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

Rearranging column order at runtime

Topic groups See also

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

A grid's *OnColumnMoved* event is fired after a column has been moved.

To prevent a user from rearranging columns at runtime, set the grid's *DragMode* property to *dmAutomatic*.

Controlling grid drawing

Topic groups See also

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

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

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

Responding to user actions at runtime

Topic groups See also

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

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

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

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

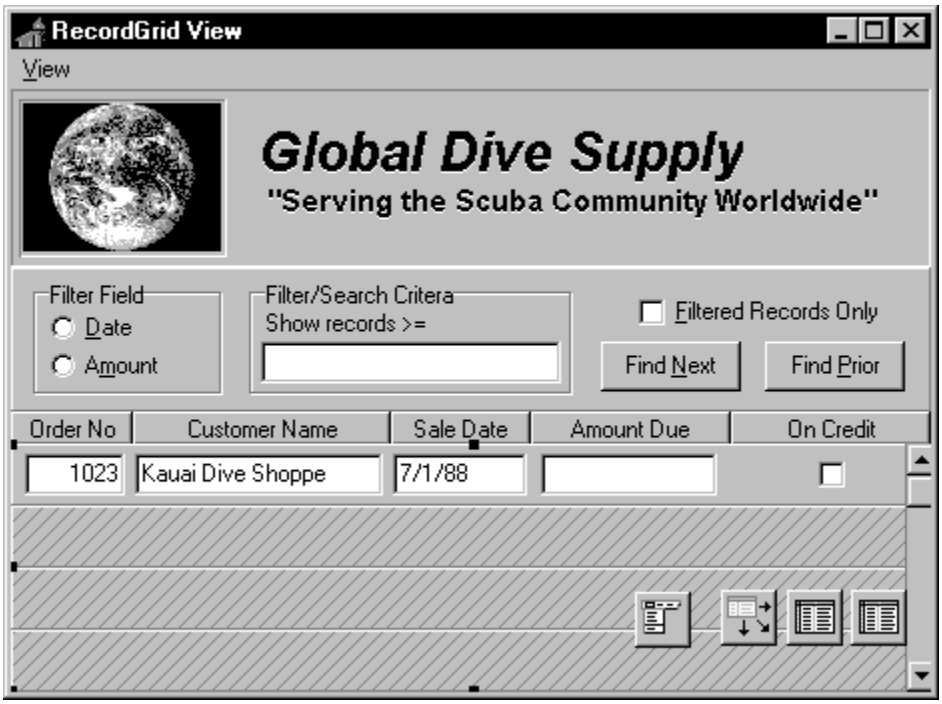
Creating a grid that contains other data-aware controls

Topic groups See also

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

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

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



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

Property	Purpose
AllowDelete	*True* (default): Permits record deletion. *false*: Prevents record deletion.
AllowInsert	*True* (default): Permits record insertion. *False*: Prevents record insertion.
ColCount	Sets the number of columns in the grid. Default = 1.
Orientation	*goVertical* (default): Display records from top to bottom. *goHorizontal*: Displays records from left to right.
PanelHeight	Sets the height for an individual panel. Default = 72.
PanelWidth	Sets the width for an individual panel. Default = 200.
RowCount	Sets the number of panels to display. Default = 3.
ShowFocus	*True* (default): Displays a focus rectangle around the current record's panel at runtime.

False: Does not display a focus rectangle.

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

Navigating and manipulating records

Topic groups See also

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.

The following figure shows the navigator that appears by default when you place it on a form at design time. The navigator consists of a series of buttons that let a user navigate from one record to another in a dataset, and edit, delete, insert, and post records. The *VisibleButtons* property of the navigator enables you to hide or show a subset of these buttons dynamically. See Choosing navigator buttons to display for more information. See Displaying fly-over help for information on associating help hints with each button. See Using a single navigator for multiple datasets for information about associating a navigator with multiple datasets.

The following table describes the buttons on the navigator.

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

Choosing navigator buttons to display

Topic groups See also

When you first place a *TDBNavigator* on a form at design time, all its buttons are visible. You can use the *VisibleButtons* property to turn off buttons you do not want to use on a form. For example, 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 following topics are discussed in this section:

- Hiding and showing navigator buttons at design time
- Hiding and showing navigator buttons at runtime

Hiding and showing navigator buttons at design time

Topic groups See also

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 time and runtime.

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

For more information about buttons and the methods they call, see the online VCL Reference.

Hiding and showing navigator buttons at runtime

Topic groups See also

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

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

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

Displaying fly-over help

Topic groups See also

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

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

Using a single navigator for multiple datasets

Topic groups See also

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

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

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

Using decision support components

Topic groups See also

The decision support components help you create cross-tabulated—or, crosstab—tables and graphs. You can then use these tables and graphs to view and summarize data from different perspectives. For more information on cross-tabulated data, see About crosstabs.The following topics are discussed in this section:

- Overview of decision support components
- Guidelines for using decision support components
- Decision support components at runtime
- Decision support components and memory control

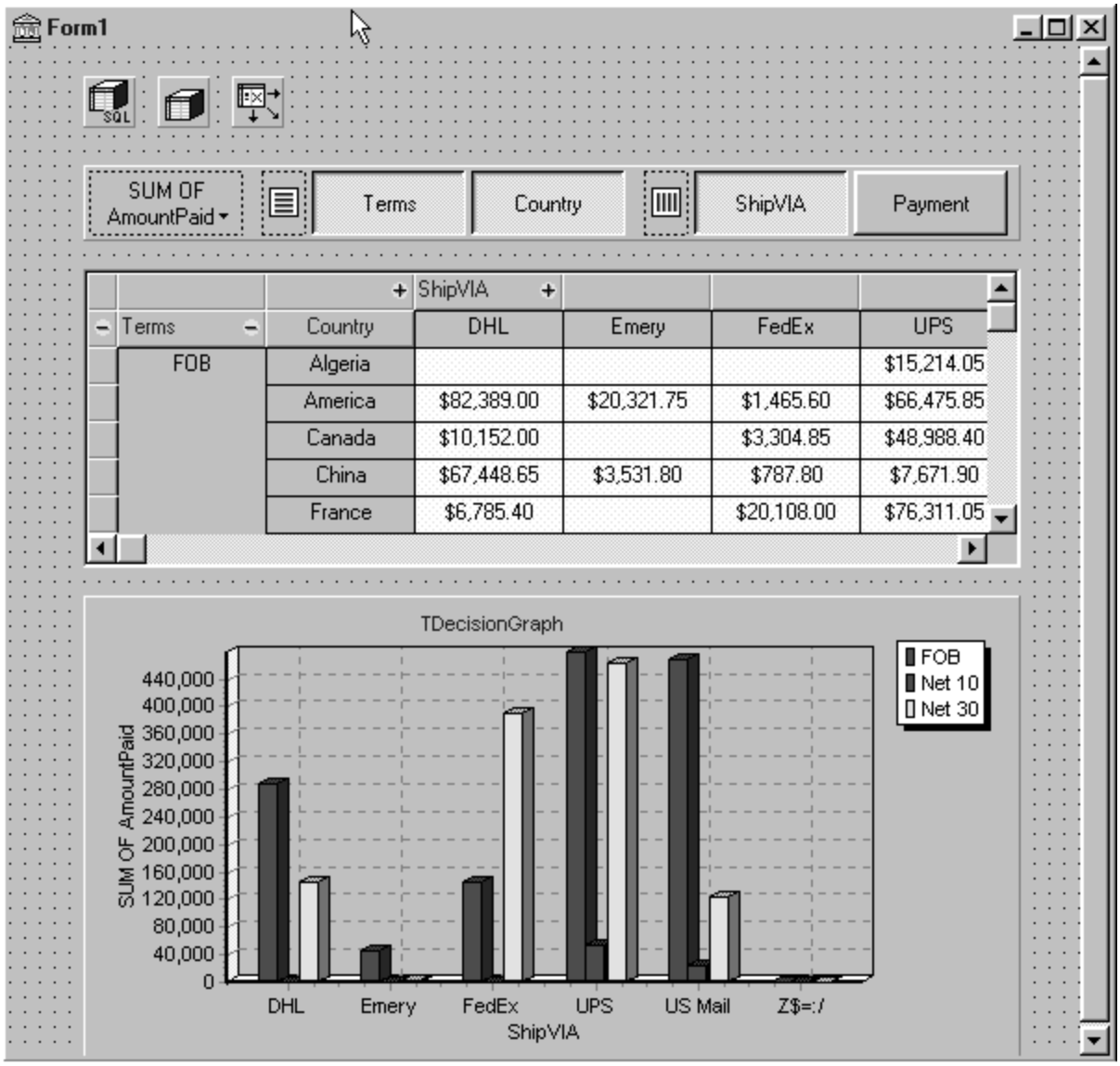
Overview of decision support components

Topic groups See also

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

- The decision cube, TDecisionCube, is a multidimensional data store. For more information see Using decision cubes.
- The decision source, TDecisionSource, defines the current pivot state of a decision grid or a decision graph. For more information, see Using decision sources.
- The decision query, TDecisionQuery, is a specialized form of *TQuery* used to define the data in a decision cube. For more information, see Using datasets with decision support components.
- The decision pivot, TDecisionPivot, lets you open or close decision cube dimensions, or fields, by pressing buttons. For more information, see Using decision pivots.
- The decision grid, TDecisionGrid, displays single- and multidimensional data in table form. For more information, see Creating and using decision grids.
- The decision graph, TDecisionGraph, displays fields from a decision grid as a dynamic graph that changes when data dimensions are modified. For more information, see Creating and using decision graphs.

The following figure shows all the decision support components placed on a form at design time.



About crosstabs

Topic groups See also

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

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

Crosstabs can be one-dimensional or multidimensional.

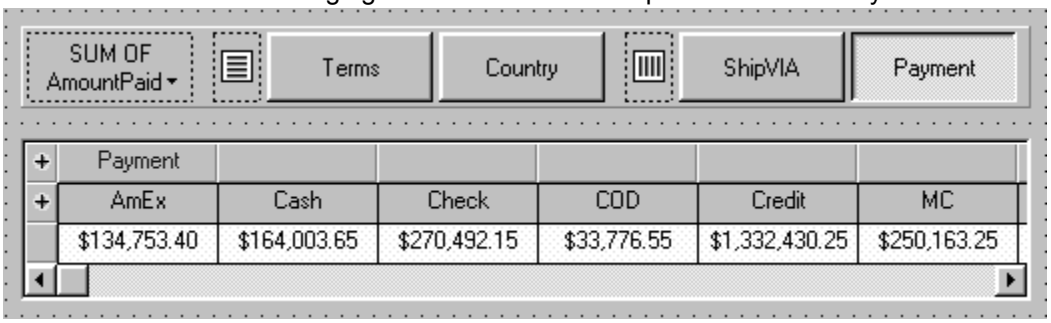
The following topics are discussed in this section:

- One-dimensional crosstabs
- Multidimensional crosstabs

One-dimensional crosstabs

Topic groups See also

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

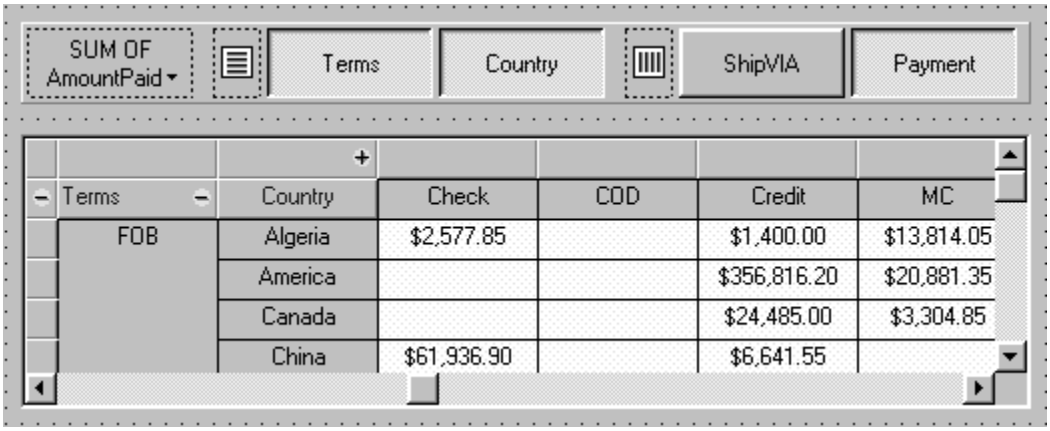


Multidimensional crosstabs

Topic groups See also

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

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



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

Guidelines for using decision support components

Topic groups See also

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

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

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

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

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

For an illustration of all decision support components on a form, see the figure Decision support components at design time.

Using datasets with decision support components

Topic groups See also

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

The decision query component, TDecisionQuery, is a specialized form of *TQuery*. You can use TDecisionQuery to more simply define the setup of dimensions (rows and columns) and summary values used to supply data to decision cubes, *TDecisionCube*. The decision query has no properties than are not inherited from other components. Important inherited properties are Active and SQL.

You can also use a TQuery or TTable component as a dataset for *TDecisionCube*, but the correct setup of the dataset and *TDecisionCube* are then the responsibility of the designer.

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

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

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

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

Creating decision datasets with TQuery or TTable

Topic groups See also

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

The SQL should look similar to this:

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

The ordering of the SELECT fields should match the ordering of the GROUP BY fields. Queries are described in more detail in Working with queries.

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

Creating decision datasets with the Decision Query editor

Topic groups See also

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

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

For instructions on using the Decision Query editor, see Using the Decision Query editor.

Using the Decision Query editor

Topic groups See also

To use the Decision Query editor:

1. Select the decision query component on the form, then right-click and choose Decision Query Editor. The Decision Query Editor dialog box appears.
2. Choose the database to use.
3. For single-table queries, click the Select Table button.

   For more complex queries involving multi-table joins, click the Query Builder button to display the SQL Builder or type the SQL statement into the edit box on the SQL tab page.
4. Return to the Decision Query Editor dialog box.
5. In the Decision Query Editor dialog box, select fields in the Available Fields list box and assign them to be either Dimensions or Summaries by clicking the appropriate right-arrow button. As you add fields to the Summaries list, select from the menu displayed the type of summary to use: sum, count, or average.
6. By default, all fields and summaries defined in the SQL property of the decision query appear in the Active Dimensions and Active Summaries list boxes. To remove a dimension or summary, select it in the list and click the left-arrow beside the list, or double-click the item to remove. To add it back, select it in the Available Fields list box and click the appropriate right-arrow.

Once you define the contents of the decision cube, you can further manipulate dimension display with its *DimensionMap* property and the buttons of *TDecisionPivot*. For more information, see Using decision cubes, Using decision sources, and Using decision pivots.

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

Using decision cubes

Topic groups See also

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

The following topics are discussed in this section:

- Decision cube properties and events
- Using the Decision Cube editor

Decision cube properties and events

Topic groups See also

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

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

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

Using the Decision Cube editor

Topic groups See also

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

The Decision Cube Editor dialog box has two tabs:

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

Viewing and changing dimension settings

Topic groups See also

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

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

Setting the maximum available dimensions and summaries

Topic groups See also

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

Viewing and changing design options

Topic groups See also

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

Using decision sources

Topic groups See also

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

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

- The ControlType property of *TDecisionSource* indicates whether the decision pivot buttons should act like check boxes (multiple selections) or radio buttons (mutually exclusive selections).
- The SparseCols and SparseRows properties of *TDecisionSource* indicate whether to display columns or rows with no values; if *True*, sparse columns or rows are displayed.
- *TDecisionSource* has the following events:
- OnLayoutChange occurs when the user performs pivots or drill-downs that reorganize the data.
- OnNewDimensions occurs when the data is completely altered, such as when the summary or dimension fields are altered.
- OnSummaryChange occurs when the current summary is changed.
- OnStateChange occurs when the Decision Cube activates or deactivates.
- OnBeforePivot occurs when changes are committed but not yet reflected in the user interface. Developers have an opportunity to make changes, for example, in capacity or pivot state, before application users see the result of their previous action.
- OnAfterPivot fires after a change in pivot state. Developers can capture information at that time.

Using decision pivots

Topic groups See also

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

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

For illustrations of decision pivots at design time, see figures in Decision support components at design time, One-dimensional crosstab, and Three-dimensional crosstab.

For information on special properties of *TDecisionPivot*, see Decision pivot properties.

Decision pivot properties

Topic groups See also

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

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

Creating and using decision grids

Topic groups See also

Decision grid components, TDecisionGrid, present cross-tabulated data in table form. These tables are also called crosstabs, described in About crosstabs. The figure Decision support components at design time shows a decision grid on a form at design time.

The following topics are discussed in this section:

- Creating decision grids
- Using decision grids
- Decision grid properties

Creating decision grids

Topic groups See also

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

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

For a description of what appears in the decision grid and how to use it, see Using decision grids..

To add a graph to the form, follow the instructions in Creating decision graphs.

Using decision grids

Topic groups See also

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

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

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

For more information about special properties and events of the decision grid, see Decision grid properties.

Opening and closing decision grid fields

Topic groups See also

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

Reorganizing rows and columns in decision grids

Topic groups See also

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

If you included a decision pivot, you can push and drag its buttons to reorganize the display. See Using decision pivots for instructions.

Drilling down for detail in decision grids

Topic groups See also

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

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

To drill down into a dimension,

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

To make the complete dimension active again,

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

Limiting dimension selection in decision grids

Topic groups See also

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

Decision grid properties

Topic groups See also

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

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

- *TDecisionGrid* has unique properties for each dimension. To set these, choose *Dimensions* in the Object Inspector, then select a dimension. Its properties then appear in the Object Inspector: *Alignment* defines the alignment of category labels for that dimension, *Caption* can be used to override the default dimension name, *Color* defines the color of category labels, *FieldName* displays the name of the active dimension, *Format* can hold any standard format for that data type, and *Subtotals* indicates whether to display subtotals for that dimension. With summary fields, these same properties are used to changed the appearance of the data that appears in the summary area of the grid. When you're through setting dimension properties, either click a component in the form or choose a component in the drop-down list box at the top of the Object Inspector.
- The *Options* property of *TDecisionGrid* lets you control display of grid lines (*cgGridLines* = *True*), enabling of outline features (collapse and expansion of dimensions with + and - indicators; *cgOutliner* = *True*), and enabling of drag-and-drop pivoting (*cgPivotable* = *True*).
- The *OnDecisionDrawCell* event of *TDecisionGrid* gives you a chance to change the appearance of each cell as it is drawn. The event passes the *String*, *Font*, and *Color* of the current cell as reference parameters. You are free to alter those parameters to achieve effects such as special colors for negative values. In addition to the *DrawState* which is passed by *TCustomGrid*, the event passes *TDecisionDrawState*, which can be used to determine what type of cell is being drawn. Further information about the cell can be fetched using the *Cells*, *CellValueArray*, or *CellDrawState* functions.
- The *OnDecisionExamineCell* event of *TDecisionGrid* lets you hook the right-click-on-event to data cells, and is intended to allow a program to display information (such as detail records) about that particular data cell. When the user right-clicks a data cell, the event is supplied with all the information which is was used to compose the data value, including the currently active summary value and a *ValueArray* of all the dimension values which were used to create the summary value.

Creating and using decision graphs

Topic groups See also

Decision graph components, TDecisionGraph, present cross-tabulated data in graphic form. Each decision graph shows the value of a single summary, such as Sum, Count, or Avg, charted for one or more dimensions. For more information on crosstabs, see One-dimensional crosstabs. For illustrations of decision graphs at design time, see the figures Decision support components at design time. and Decision graphs bound to different decision sources.

The following topics are discussed in this section:

- Creating decision graphs
- Using decision graphs
- The decision graph display
- Customizing decision graphs

Creating decision graphs

Topic groups See also

To create a form with one or more decision graphs,

1. Follow steps 1–3 listed under Guidelines for using decision support components.
2. Add one or more decision graph components (TDecisionGraph) and bind them to the decision source, TDecisionSource, with the Object Inspector by setting their *DecisionSource* property to the appropriate decision source component.
3. Continue with steps 5–7 listed under Guidelines for using decision support components.
4. Finally, right-click the graph and choose Edit Chart to modify the appearance of the graph series. You can set template properties for each graph dimension, then set individual series properties to override these defaults. For details, see Customizing decision graphs.

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

To add a decision grid—or crosstab table—to the form, follow the instructions in Creating and using decision grids.

Using decision graphs

Topic groups See also

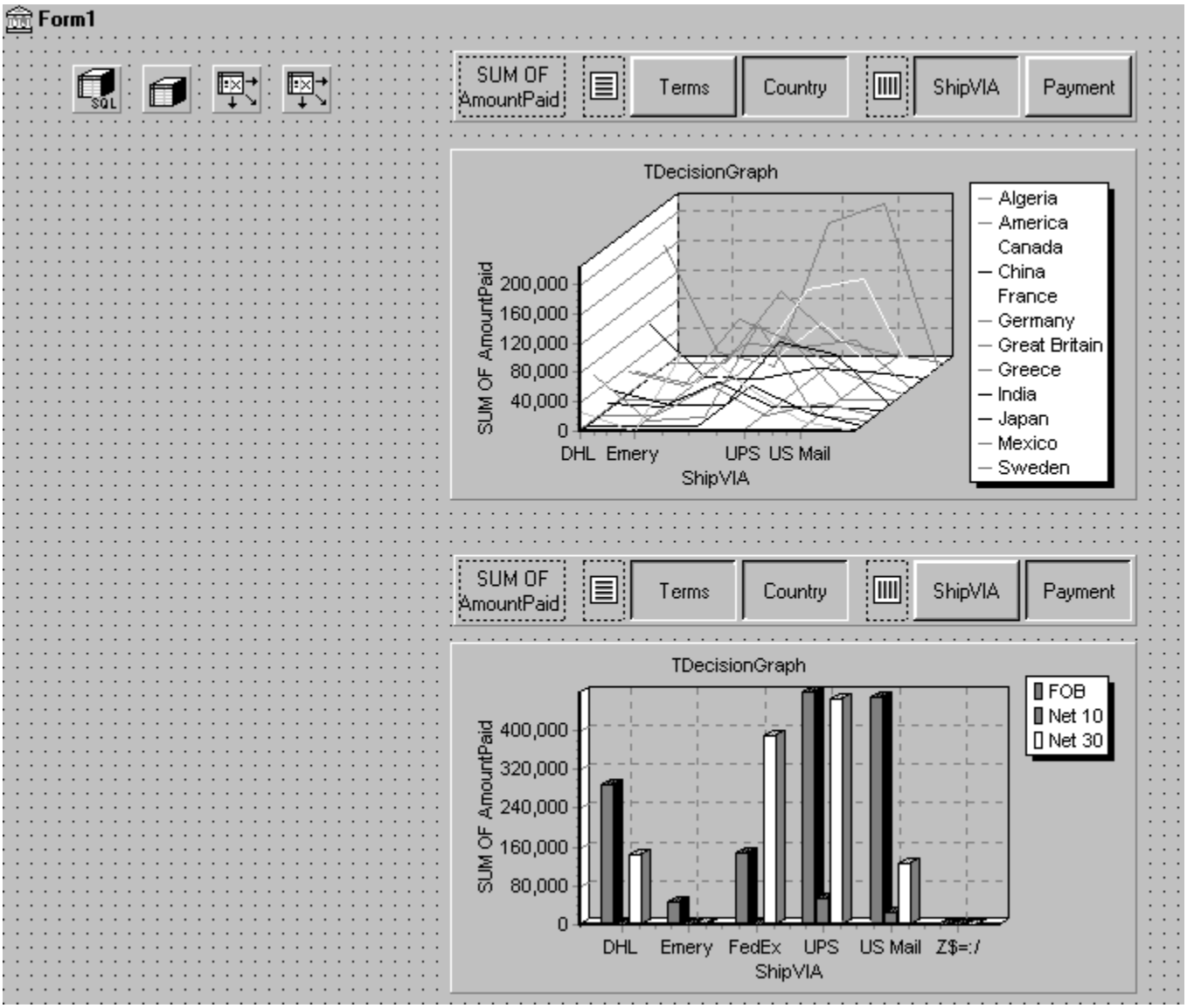
The decision graph component, TDecisionGraph, displays fields from the decision source (TDecisionSource) as a dynamic graph that changes when data dimensions are opened, closed, dragged and dropped, or rearranged with the decision pivot (TDecisionPivot).

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

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

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

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



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

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

For a discussion of decision graph properties and how to change the appearance and behavior of decision graphs, see Customizing decision graphs.

The decision graph display

Topic groups See also

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

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

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

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

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

Customizing decision graphs

Topic groups See also

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

To customize a graph,

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

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

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

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

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

To customize individual series, follow the instructions in Customizing decision graph series.

Setting decision graph template defaults

Topic groups See also

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

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

You can

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

Changing the default decision graph type

Topic groups See also

To change the default graph type,

1. Select a template in the Series list on the Chart page of the Chart Editing dialog box.
2. Click the Change button.
3. Select a new type and close the Gallery dialog box.

Changing other decision graph template properties

Topic groups See also

To change color or other properties of a template,

1. Select the Series page at the top of the Chart Editing dialog box.
2. Choose a template in the drop-down list at the top of the page.
3. Choose the appropriate property tab and select settings.

Viewing overall decision graph properties

Topic groups See also

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

1. Select the Chart page at the top of the Chart Editing dialog box.
2. Choose the appropriate property tab and select settings.

Customizing decision graph series

Topic groups

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

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

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

To define series templates and set overall graph defaults, see Setting decision graph template defaults.

Changing the series graph type

Topic groups See also

By default, each series has the same graph type, defined by the template for its dimension. To change all series to the same graph type, you can change the template type. See Changing other decision graph series properties. for instructions.

To change the graph type for a single series,

1. Select a series in the Series list on the Chart page of the Chart Editor.
2. Click the Change button.
3. Select a new type and close the Gallery dialog box.
4. Check the Save Series check box.

Changing other decision graph series properties

Topic groups See also

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

1. Select the Series page at the top of the Chart Editing dialog box.
2. Choose a series in the drop-down list at the top of the page.
3. Choose the appropriate property tab and select settings.
4. Check the Save Series check box.

Saving decision graph series settings

Topic groups See also

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

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

Decision support components at runtime

Topic groups See also

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

- Decision pivots at runtime
- Decision grids at runtime
- Decision graphs at runtime

Decision pivots at runtime

Topic groups See also

Users can:

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

Decision grids at runtime

Topic groups See also

Users can:

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

Decision graphs at runtime

Topic groups See also

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

Decision support components and memory control

Topic groups See also

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

Memory consumption can be limited by the following techniques:

- Setting maximum dimensions, summaries, and cells
- Setting dimension state
- Using paged dimensions

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

Setting maximum dimensions, summaries, and cells

Topic groups See also

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

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

Setting dimension state

Topic groups See also

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

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

Using paged dimensions

Topic groups See also

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

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

Related topic groups

Developing Database Applications

- Database user interface
- Database architecture
- Database design
- Database features
- Database models
- Database types
- InterBase Express
- Building ADO applications
- BDE architecture
- Flat file applications
- Mixed models
- MIDAS-based multi-tiered application
- Creating the client application
- Customizing multi-tiered applications
- Creating the application server
- Multitiered applications
- Web-based multitiered applications
- Using provider components
- Sessions
- Connecting to databases
- Datasets
- Fields
- Tables
- Queries
- Stored procedures
- ADO components
- Client datasets
- Cached updates
- Database controls
- Decision support

Database user interface

Related topic groups

- Displaying a single record
- Displaying multiple records
- Analyzing data
- Selecting what data to show
- Writing reports

Database architecture

Related topic groups

- Database architecture
- Planning for scalability

Database design

Related topic groups

- Designing the user interface

Database features

Related topic groups

- Database security
- Transactions
- The Data Dictionary
- Referential integrity, stored procedures, and triggers

Database models

Related topic groups

- Single-tiered database applications
- Two-tiered database applications
- Understanding the architecture of a multi-tiered application

Database types

Related topic groups

- Local databases
- Remote database servers

InterBase Express

Related topic groups

- InterBase Express

Building ADO applications

Related topic groups

- ADO-based applications
- ADO-based architecture
- Understanding ADO databases and datasets
- Connecting to ADO databases
- Retrieving data
- Creating and restructuring ADO database tables

BDE architecture

Related topic groups

- Understanding databases and datasets
- Using sessions
- Connecting to databases
- Using transactions
- Explicitly controlling transactions
- Using a database component for transactions
- Using the TransIsolation property
- Using passthrough SQL
- Using local transactions
- Caching updates
- Creating and restructuring database tables

Flat file applications

Related topic groups

- Creating a new dataset using persistent fields
- Creating a dataset using field and index definitions
- Creating a dataset based on an existing table

Mixed models

Related topic groups

- Supporting the briefcase model
- Scaling up to a three-tiered application

MIDAS-based multi-tiered application

Related topic groups

- Overview of a MIDAS-based multi-tiered application
- The structure of the client application
- The structure of the application server
- Using MTS
- Pooling remote data modules
- Using the IAppServer interface
- Choosing a connection protocol
- Using DCOM connections
- Using Socket connections
- Using Web connections
- Using OLEnterprise
- Using CORBA connections

Creating the client application

Related topic groups

- Creating the client application
- Connecting to the application server
- Specifying a connection using DCOM
- Specifying a connection using sockets
- Specifying a connection using HTTP
- Specifying a connection using OLEnterprise
- Specifying a connection using CORBA
- Brokering connections
- Managing server connections
- Connecting to the server
- Dropping or changing a server connection
- Calling server interfaces

Customizing multi-tiered applications

Related topic groups

- Extending the interface of the application server
- Supporting state information in remote data modules

Creating the application server

Related topic groups

- Creating the application server
- Setting up the remote data module
- Configuring TRemoteDataModule
- Configuring TMTSDataModule
- Configuring TCORBADataModule
- Creating a data provider for the application server

Multitiered applications

Related topic groups

- Creating multi-tiered applications: Overview
- Advantages of the multi-tiered database model
- Understanding MIDAS technology
- Building a multi-tiered application
- Managing transactions in multi-tiered applications
- Supporting master/detail relationships

Web-based multitiered applications

Related topic groups

- Writing MIDAS Web applications
- Distributing a client application as an ActiveX control
- Creating an Active Form for the client application
- Building Web applications using InternetExpress
- Building an InternetExpress application
- Using the javascript libraries
- Granting permission to access and launch the application server
- Using an XML broker
- Creating Web pages with a MIDAS page producer
- Using the Web page editor
- Setting Web item properties
- Customizing the MIDAS page producer template

Using provider components

Related topic groups

- Using provider components
- Determining the source of data
- Choosing how to apply updates
- Controlling what information is included in data packets
- Specifying what fields appear in data packets
- Setting options that influence the data packets
- Adding custom information to data packets
- Responding to client data requests
- Responding to client update requests
- Editing delta packets before updating the database
- Influencing how updates are applied
- Screening individual updates
- Resolving update errors on the provider
- Applying updates to datasets that do not represent a single table
- Responding to client-generated events
- Handling server constraints

Sessions

Related topic groups

- Managing database sessions: Overview
- Working with a session component
- Using the default session
- Creating additional sessions
- Naming a session
- Activating a session
- Customizing session start-up
- Specifying default database connection behavior
- Creating, opening, and closing database connections
- Closing a single database connection
- Closing all database connections
- Dropping temporary database connections
- Searching for a database connection
- Retrieving information about a session
- Working with BDE aliases
- Specifying alias visibility
- Making session aliases visible to other sessions and applications
- Determining known aliases, drivers, and parameters
- Creating, modifying, and deleting aliases
- Iterating through a session's database components
- Specifying Paradox directory locations
- Specifying the control file location
- Specifying a temporary files location
- Working with password-protected Paradox tables
- Using the AddPassword method
- Using the RemovePassword and RemoveAllPasswords methods
- Using the GetPassword method and OnPassword event
- Managing multiple sessions
- Using a session component in data modules

Connecting to databases

Related topic groups

- Connecting to databases: Overview
- Understanding persistent and temporary database components
- Using temporary database components
- Creating database components at design time
- Creating database components at runtime
- Controlling connections
- Associating a database component with a session
- Specifying a BDE alias
- Setting BDE alias parameters
- Controlling server login
- Connecting to a database server
- Special considerations when connecting to a remote server
- Working with network protocols
- Using ODBC
- Disconnecting from a database server
- Closing datasets without disconnecting from a server
- Iterating through a database component's datasets
- Understanding database and session component interactions
- Using database components in data modules
- Executing SQL statements from a TDatabase component
- Executing SQL statements without result sets
- Executing SQL statements with result sets
- Executing parameterized SQL statements

Datasets

Related topic groups

- Understanding datasets: Overview
- What is TDataSet?
- Types of datasets
- Opening and closing datasets
- Determining and setting dataset states
- Inactivating a dataset
- Browsing a dataset
- Enabling dataset editing
- Enabling insertion of new records
- Enabling index-based searches and ranges on tables
- Calculating fields
- Filtering records
- Updating records
- Navigating datasets
- Using the First and Last methods
- Using the Next and Prior methods
- Using the MoveBy method
- Using the Eof and Bof properties
- Eof
- Bof
- Marking and returning to records
- Searching datasets
- Using Locate
- Using Lookup
- Displaying and editing a subset of data using filters
- Enabling and disabling filtering
- Creating filters
- Setting the Filter property
- Writing an OnFilterRecord event handler
- Setting filter options
- Navigating records in a filtered dataset
- Modifying data
- Editing records
- Adding new records
- Inserting records
- Appending records
- Deleting records
- Posting data to the database
- Canceling changes
- Modifying entire records
- Using dataset events

- Aborting a method
- Using OnCalcFields
- Using BDE-enabled datasets
- Overview of BDE-enablement
- Handling database and session connections
- Using the DatabaseName and SessionName properties
- Working with BDE handle properties
- Working with cached updates
- Caching BLOBs

Fields

Related topic groups

- Working with field components: Overview
- Understanding field components
- Dynamic field components
- Persistent field components
- Creating persistent fields
- Arranging persistent fields
- Defining new persistent fields

Tables

Related topic groups

- Working with tables: Overview
- Using table components
- Setting up a table component
- Specifying a database location
- Specifying a table name
- Specifying the table type for local tables
- Opening and closing a table
- Controlling read/write access to a table
- Searching for records
- Searching for records based on indexed fields
- Executing a search with Goto methods
- Executing a search with Find methods
- Specifying the current record after a successful search
- Searching on partial keys
- Searching on alternate indexes
- Sorting records
- Retrieving a list of available indexes with GetIndexNames
- Specifying an alternative index with IndexName
- Specifying a dBASE index file
- Specifying sort order for SQL tables
- Specifying fields with IndexFieldNames
- Examining the field list for an index
- Working with a subset of data
- Understanding the differences between ranges and filters
- Creating and applying a new range
- Setting the beginning of a range
- Setting the end of a range
- Setting start- and end-range values
- Specifying a range based on partial keys
- Including or excluding records that match boundary values
- Applying a range
- Canceling a range
- Modifying a range
- Editing the start of a range
- Editing the end of a range
- Deleting all records in a table
- Deleting a table
- Renaming a table
- Creating a table
- Importing data from another table
- Using TBatchMove

- Creating a batch move component
- Specifying a batch move mode
- Appending
- Updating
- Appending and updating
- Copying
- Deleting
- Mapping data types
- Executing a batch move
- Handling batch move errors
- Synchronizing tables linked to the same database table
- Creating master/detail forms
- Building an example master/detail form
- Working with nested tables

Queries

Related topic groups

- Working with queries: Overview
- Using queries effectively
- Queries for desktop developers
- Queries for server developers
- What databases can you access with a query component?
- Using a query component: an overview
- Specifying the SQL statement to execute
- Specifying the SQL property at design time
- Specifying an SQL statement at run time
- Setting the SQL property directly
- Loading the SQL property from a file
- Loading the SQL property from string list object
- Setting parameters
- Supplying parameters at design time
- Supplying parameters at run time
- Using a data source to bind parameters
- Executing a query
- Executing a query at design time
- Executing a query at runtime
- Executing a query that returns a result set
- Executing a query without a result set
- Preparing a query
- Unpreparing a query to release resources
- Creating heterogenous queries
- Improving query performance
- Disabling bi-directional cursors
- Working with result sets
- Enabling editing of a result set
- Local SQL requirements for a live result set
- Restrictions on live queries
- Remote server SQL requirements for a live result set
- Restrictions on updating a live result set
- Updating a read-only result set

Stored procedures

Related topic groups

- Working with stored procedures: Overview
- When should you use stored procedures?
- Using a stored procedure
- Creating a stored procedure component
- Creating a stored procedure
- Preparing and executing a stored procedure
- Stored procedures that return result sets
- Retrieving a result set with a TQuery
- Retrieving a result set with a TStoredProc
- Stored procedures that return data using parameters
- Retrieving individual values with a TQuery
- Retrieving individual values with a TStoredProc
- Stored procedures that perform actions on data
- Executing an action stored procedure with a TQuery
- Executing an action stored procedure with a TStoredProc
- Understanding stored procedure parameters
- Using input parameters
- Using output parameters
- Using input/output parameters
- Using the result parameter
- Accessing parameters at design time
- Setting parameter information at design time
- Creating parameters at runtime
- Binding parameters
- Viewing parameter information at design time
- Working with Oracle overloaded stored procedures

ADO components

Related topic groups

- Working with ADO components
- Overview of ADO components
- Connecting to ADO data stores
- Connecting to a data store
- Using a TADOConnection versus a dataset's ConnectionString
- Specifying the connection
- Accessing the connection object
- Activating and deactivating the connection
- Determining what a connection component is doing
- Fine-tuning a connection
- Specifying connection attributes
- Controlling timeouts
- Controlling the connection login
- Working with a connection's dataset components
- Accessing the connection's datasets
- Accessing the connection's commands
- Listing available tables
- Listing available stored procedures
- Working with (connection) transactions
- Using transaction methods
- Using transaction events
- Using ADO datasets
- Features common to all ADO dataset components
- Modifying data
- Navigating in a dataset
- Using visual data-aware controls
- Connecting to a data store
- Working with recordsets
- Using batch updates
- Opening the dataset in batch update mode
- Inspecting the update status of individual rows
- Filtering multiple rows based on update status
- Applying the batch updates to base tables
- Canceling batch updates
- Loading data from and saving data to files
- Using parameters in commands
- Using TADODataSet
- Retrieving a dataset using a command
- Using TADOTable
- Specifying the table to use
- Using TADOQuery

- Specifying SQL statements
- Executing SQL statements
- Using TADOStoredProc
- Specifying the stored procedure
- Executing the stored procedure
- Using parameters with stored procedures
- Using TADOStoredProc input parameters
- Using TADOStoredProc output parameters
- Using TADOStoredProc return value parameters
- Executing commands
- Specifying the command
- Using the Execute method
- Canceling commands
- Retrieving result sets with commands
- Handling command parameters

Client datasets

Related topic groups

- Creating and using a client dataset: Overview
- Working with data using a client dataset
- Navigating data in client datasets
- Limiting what records appear
- Representing master/detail relationships
- Constraining data values
- Making data read-only
- Editing data
- Undoing changes
- Saving changes
- Sorting and indexing
- Adding a new index
- Deleting and switching indexes
- Using indexes to group data
- Indexing on the fly
- Representing calculated values
- Using internally calculated fields in client datasets
- Using maintained aggregates
- Specifying aggregates
- Aggregating over groups of records
- Obtaining aggregate values
- Adding application-specific information to the data
- Copying data from another dataset
- Assigning data directly
- Cloning a client dataset cursor
- Using a client dataset with a data provider
- Specifying a data provider
- Getting parameters from the application server
- Passing parameters to the application server
- Sending query or stored procedure parameters
- Limiting records with parameters
- Overriding the dataset on the application server
- Requesting data from an application server
- Handling constraints
- Handling constraints from the server
- Adding custom constraints
- Updating records
- Applying updates
- Reconciling update errors
- Refreshing records
- Communicating with providers using custom events

- Using a client dataset with a data provider
- Creating a new dataset
- Loading data from a file or stream
- Merging changes into Data
- Saving data to a file or stream

Cached updates

Related topic groups

- Working with cached updates: Overview
- Deciding when to use cached updates
- Using cached updates
- Enabling and disabling cached updates
- Fetching records
- Applying cached updates
- Applying cached updates with a database component method
- Applying cached updates with dataset component methods
- Applying updates for master/detail tables
- Canceling pending cached updates
- Canceling pending updates and disabling further cached updates
- Canceling pending cached updates
- Canceling updates to the current record
- Undeleting cached records
- Specifying visible records in the cache
- Checking update status
- Using update objects to update a dataset
- Specifying the UpdateObject property for a dataset
- Using a single update object
- Using multiple update objects
- Creating SQL statements for update components
- Creating SQL statements at design time
- Understanding parameter substitution in update SQL statements
- Composing update SQL statements
- Accessing an update component's Query property
- Using the DeleteSQL, InsertSQL, and ModifySQL properties
- Executing update statements
- Calling the Apply method
- Calling the SetParams method
- Executing an update statement
- Using dataset components to update a dataset
- Updating a read-only result set
- Controlling the update process
- Determining if you need to control the updating process
- Creating an OnUpdateRecord event handler
- Handling cached update errors
- Referencing the dataset to which to apply updates
- Indicating the type of update that generated an error
- Specifying the action to take
- Working with error message text
- Accessing a field's OldValue, NewValue, and CurValue properties

Database controls

Related topic groups

- Using data controls
- Using common data control features
- Associating a data control with a dataset
- Editing and updating data
- Enabling editing in controls on user entry
- Editing data in a control
- Disabling and enabling data display
- Refreshing data
- Enabling mouse, keyboard, and timer events
- Using data sources
- Using TDataSource properties
- Setting the DataSet property
- Setting the Name property
- Setting the Enabled property
- Setting the AutoEdit property
- Using TDataSource events
- Using the OnDataChange event
- Using the OnUpdateData event
- Using the OnStateChange event
- Controls that represent a single field
- Displaying data as labels
- Displaying and editing fields in an edit box
- Displaying and editing text in a memo control
- Displaying and editing text in a rich edit memo control
- Displaying and editing graphics fields in an image control
- Displaying and editing data in list and combo boxes
- Displaying and editing data in a list box
- Displaying and editing data in a combo box
- Displaying and editing data in lookup list and combo boxes
- Specifying a list based on a lookup field
- Specifying a list based on a secondary data source
- Setting lookup list and combo box properties
- Searching incrementally for list item values
- Handling Boolean field values with check boxes
- Restricting field values with radio controls
- Viewing and editing data with TDBGrid
- Using a grid control in its default state
- Creating a customized grid
- Understanding persistent columns
- Determining the source of a column property at runtime
- Creating persistent columns

- Deleting persistent columns
- Arranging the order of persistent columns
- Defining a lookup list column
- Defining a pick list column
- Putting a button in a column
- Setting column properties at design time
- Restoring default values to a column
- Displaying ADT and array fields
- Setting grid options
- Editing in the grid
- Rearranging column order at design time
- Rearranging column order at runtime
- Controlling grid drawing
- Responding to user actions at runtime
- Creating a grid that contains other data-aware controls
- Navigating and manipulating records
- Choosing navigator buttons to display
- Hiding and showing navigator buttons at design time
- Hiding and showing navigator buttons at runtime
- Displaying fly-over help
- Using a single navigator for multiple datasets

Decision support

Related topic groups

- Using decision support components
- Overview of decision support components
- About crosstabs
- One-dimensional crosstabs
- Multidimensional crosstabs
- Guidelines for using decision support components
- Using datasets with decision support components
- Creating decision datasets with TQuery or TTable
- Creating decision datasets with the Decision Query editor
- Using the Decision Query editor
- Using decision cubes
- Decision cube properties and events
- Using the Decision Cube editor
- Viewing and changing dimension settings
- Setting the maximum available dimensions and summaries
- Viewing and changing design options
- Using decision sources
- Using decision pivots
- Decision pivot properties
- Creating and using decision grids
- Creating decision grids
- Using decision grids
- Opening and closing decision grid fields
- Reorganizing rows and columns in decision grids
- Drilling down for detail in decision grids
- Limiting dimension selection in decision grids
- Decision grid properties
- Creating and using decision graphs
- Creating decision graphs
- Using decision graphs
- The decision graph display
- Customizing decision graphs
- Setting decision graph template defaults
- Changing the default decision graph type
- Changing other decision graph template properties
- Viewing overall decision graph properties
- Customizing decision graph series
- Changing the series graph type
- Changing other decision graph series properties
- Saving decision graph series settings
- Decision support components at runtime

- Decision pivots:runtime behavior
- Decision grids at runtime
- Decision graphs at runtime
- Decision support components and memory control
- Setting maximum dimensions, summaries, and cells
- Setting dimension state
- Using paged dimensions

Link not found

The topic you requested is either not available or not linked to this Help system. This can occur if you launched this Help file from a system on which Delphi has not yet been installed, or if the subject matter you are requesting is not available in your edition of Delphi.

Borland INPRISE

The topic you requested is now loading. If it does not appear within a few seconds, the topic is either not available or not linked to this Help system. This can occur if you launched this Help file from a system on which Delphi has not yet been installed, or if the subject matter you are requesting is not available in your edition of Delphi.