Developing Client/Server Applications

CHAPTER

29

IN THIS CHAPTER

So what's all this hoopla about "client/server"? It seems that everyone these days is either using or developing some sort of client/server or enterprise system. Unless you've invested the time to understand what client/server is all about, it's easy to become confused over what exactly client/server is and what it offers you that other technologies do not.

If you're a Delphi developer, it wouldn't be a surprise if you've been overwhelmed with all this client/server rhetoric. Delphi 5 is, after all, a client/server development environment. However, that doesn't mean that everything you develop with Delphi is client/server. Nor does it mean that just because an application accesses data from a client/server database, such as Oracle, Microsoft SQL, or InterBase, that it's a client/server application.

This chapter discusses the elements that make up a client/server system. It compares client/server development to traditional desktop and mainframe database development. It also illustrates reasons for using a client/server solution. It discusses how Delphi 5 provides the capability to develop client/server (three-tier) applications. This chapter also points out some pitfalls desktop database developers fall into when moving to client/server.

Why Client/Server?

A typical example of when you might consider a client/server solution would be the following: Imagine that you're responsible for a departmental-level application that accesses data residing on a LAN or file server. Various people within your department may use this application. As this data becomes of greater use to your department, other applications are created to make use of this data.

Suppose this data becomes of interest to other departments within your company. Now, additional applications will have to be built for these departments. This may also require you to move the data to a database server to make it more globally available. As the data becomes of greater interest company-wide, decision-makers must be able to access it through a means that not only gets them the data quickly but also presents the data such that it actually helps in the decision-making process.

The global availability of this data creates several problems inherent in desktop database access across network connections. Two of these problems may be excessive network traffic (creating a bottleneck in data retrieval) and data security.

This is a simplified example, yet it does illustrate a situation in which you might consider the need for a client/server solution. A client/server solution would provide the following features:

- Allow departmental access to the data, enabling departments to process only the part of the business for which they're responsible
- Provide data access to decision-makers efficiently in the way the data should be presented
- Enhance centralized control by MIS of maintaining data integrity while placing less emphasis on centralized control of data analysis and use

- Enforce data integrity rules for the entire database
- Provide better division of labor between the client and the server (each performs the tasks for which it's best suited)
- Be able to use the advanced data integrity capabilities provided by most database servers
- Reduce network traffic because subsets of data are returned to the client, as opposed to entire tables as is the case with desktop databases

Keep in mind that this list is not all-inclusive. As you get into the rest of the chapter, you'll see additional benefits to moving to a client/server-based system.

It's also necessary to mention that making the move to client/server isn't always the right thing to do. As a developer, you must ensure that you've performed a thorough analysis of your user requirements to determine whether client/server is what you need. One consideration you must take into account is that client/server systems are costly. This cost includes network software, the server OS, the database server, and hardware capable of housing such software. Additionally, there will be a significant learning period if users are unfamiliar with the server OS and database server software.

Client/Server Architecture

The typical client/server architecture is one in which the front end (or end user—the *client*) accesses and processes data from a remote machine (the *server*). There is no "true" definition of client/server. However, you can think of it as if the server provides a service and the client requests a service from the server. There may be many clients that request such services from the same server. It's up to the server to decide how to process such requests. Also, there may be more than just the client and server to a client/server system. We'll discuss this further in the section covering three-tier systems.

In a client/server environment, the server handles much more than just data distribution. In fact, the server, more than likely, performs the bulk of the business logic. It also governs how the data is to be accessed and manipulated by the client. The client applications really only serve as a means to present data to or extract data from the end user. The following subsections explain in more detail the responsibilities of the client and the server. Additionally, we'll talk about *business rules*, which are the governing rules for client access to server data.

The Client

The client can be either a GUI or non-GUI application. Delphi 5 allows you to develop both the client and any middle-layer application servers in three-tier models. The database server is most likely developed using an RDBMS such as Oracle or InterBase.

Client applications provide the interface for users needing to manipulate data on the server end. It's through the client that services are requested of the server. A typical service might be, for example, adding a customer, adding an order, or printing a report. Here, the client simply makes the request and provides any necessary data. The server carries the responsibility of processing the request. This doesn't mean that the client is not capable of performing any of the logic. It's entirely possible that the client can carry out most, if not all, of the business logic in the entire application. In this case, this is what we refer to as a *fat client*.

Scalable Applications

You'll often hear the term *scalability* in reference to client/server development with Delphi. What exactly is scalability? Well, to some it means the ability to easily access server databases using Delphi's powerful database features. Also, it can mean to rapidly increase the number of users and demands on a system with minimal or no effect on performance. To others, it means magically turning a desktop application into a client application by simply changing an alias in the application. Unfortunately, the latter is not really true. Sure, you can change an `Alias` property and suddenly access data from a server database. However, this doesn't turn your application into a client application nor does is scale your application. A key advantage to client/server is that you can take advantage of the powerful features offered by the server. It would be impossible to take advantages of these features if your application is designed using desktop database methods.

The Server

The server provides the services to the client. It essentially waits for the client to make a request and then processes that request. A server must be capable of processing multiple requests from multiple clients and also must be capable of prioritizing these requests. More than likely, the server will run continuously to allow constant access to its services.

NOTE

A client doesn't necessarily have to reside on a different machine from the server. Often, the background tasks performed on the data may well reside on the same server machine.

Business Rules

What exactly are business rules? In short, business rules are the governing procedures that dictate how clients access data on the server. These rules are implemented in programming code

on the client, the server, or both. In Delphi 5, business rules are implemented in the form of Object Pascal code. On the server side, business rules are implemented in the form of SQL stored procedures, triggers, and other database objects native to server databases. In three-tier models, business rules can be implemented in the middle tier. We'll discuss these objects later in the chapter.

It's important that you understand that business rules define how the entire system will behave. Without business rules, you have nothing more than data residing on one machine and a GUI application on another and no method for connecting the two.

At some point in the design phase of developing your system, you must decide what processes must exist in your system. Take an inventory system, for example. Here, typical processes would be tasks such as placing an order, printing an invoice, adding a customer, ordering a part, and so on. As stated earlier, these rules are implemented in Object Pascal code on the client or on a middle tier. These rules may also be SQL code on the server or a combination of all three. When the majority of rules are placed on the server, we refer to this as a *fat server*. When most of the rules exist on the client, this is called a *fat client*. When the rules exist on the middle tier, we can still refer to this as a fat server as well. How much and what type of control is required over the data determine what side the business rules should exist on.

NOTE

You'll often see *three-tier* referred to as n-*tier* or *multitier*. The terms n-*tier* and *multitier* are each misnomers. In a three-tier model, you typically have one or more clients, business logic, and the database server. The business logic may very well be distributed into many pieces on several different machines or even application servers. It starts to seem a bit absurd when you start to refer to this as a 10-, 15-, or even 25-tier system. We prefer to think of the business logic (or middle) tier as a single tier regardless of how many boxes and application servers it requires.

Fat Client, Fat Server, or Middle Tier: Where Do Business Rules Belong?

The decision about where you want the business rules to exist, or how you want to separate business rules between the server and clients, depends on several factors. Some of these factors may include data security, data integrity, centralized control, and proper distribution of work.

Data Security Concerns

Security concerns come into play when you want to provide limited access to various parts of the data or to various tasks that may be performed on the data. This is done through user-access privileges to various database objects such as views and stored procedures. We'll

discuss these objects later in this chapter. By using access privileges to database objects, you can restrict a user's access to only those parts of the data that he or she needs. Privileges and stored procedures exist on the server.

One very important concept to remember is that client/server databases are designed so that a wide range of client applications and tools can access them. Although you may have limited access to data as defined in the coding logic of your client application, nothing prevents a user from using another tool to view or edit tables within your database. By making database access accessible only through views and stored procedures, you can prevent unauthorized access to your data. This also plays an important role in maintaining data integrity, as discussed in the next section.

Data Integrity Concerns

Data integrity refers to the correctness and completeness of the data on the server. Unless you take the necessary measures to protect the data, it's possible that this data may get corrupted. Examples of data corruption are placing an order on a nonexistent or depleted product, changing the quantity of a product on an order without adjusting the cost, or deleting a customer with an outstanding balance.

So how do you protect data integrity? One way is to limit the type of operations that can be performed on the data through stored procedures. Another way is by placing the bulk of the business logic on the server or on the middle layer. For example, suppose that in an inventory system, you have a client application that contains most of the business logic. In the client application, the procedure to delete a customer might be smart enough to look at the server data to determine whether a customer has an outstanding balance. This is fine for the client application. However, because this logic exists only with the client and not with the server, there's nothing to prevent a user from loading Database Desktop or some other client tool and deleting a customer directly from the table. To prevent this, you revoke access to the customer table to all users. You then provide a stored procedure on the server that takes care of deleting the user but only after making the necessary checks. Because nobody has access to the tables directly, all users are forced to use the stored procedure.

This is only one way that a business rule existing on the server can protect data integrity. The same thing can be accomplished by placing the necessary checks in triggers or by providing views to only the data the users need access to. It's important to remember that data on the server is there so that many departments through different applications can access it. The more business rules that exist on the server, the more control you have over protecting the data.

Centralized Control of Data

Another benefit to having the business logic on the server, or on another layer in a three-tier setup, is that MIS can implement updates to this business logic without affecting the operation

of the client applications. That means that if additional code were to be added to any stored procedures, this change is transparent to the clients as long as the client interfaces to the server aren't affected by the change. This makes life for MIS much easier and benefits the company overall because MIS can do its job better.

Work Distribution

By placing business rules on the server, or by separating them on various middle tiers, MIS can more easily perform the tasks of dividing up responsibilities to specific departments and still maintain the integrity/security of the server data. This allows departments to share the same data yet manipulate only that data necessary to accomplish their particular objectives. This distribution of work is accomplished by granting access to only those stored procedures and other database objects necessary for a particular department.

As an example, we'll use the inventory system again. To be more specific, let's say this is an inventory system for an automotive parts warehouse. Here, several people need to access the same data but for different purposes. A cashier must be able to process invoices, add and remove customers, and change customer information. Warehouse personnel must be able to add new parts to the database as well as order new parts. Accounting personnel must be able to perform their part of the system as well. It's not likely that warehouse personnel will have to run a monthly budget report. Nor is it likely that accounting personnel will have to change customer address information. By creating these business rules on the server, it's possible to grant access based on the needs of a person and/or department. Here, cashier personnel will have access to customer/invoice rules. Warehouse personnel will have access to business rules specific to their needs, whereas accounting personnel can access accounting-related data.

Distribution of work refers not only to dividing up work among various clients but also to determining what work would best be performed on a client as opposed to the server or middle layers. As a developer, you must evaluate various strategies that might allow you to assign CPU-intensive operations to the fast client machines, thus relieving the server so that it can perform less intensive operations. Of course, in deciding which strategies to employ, you must also consider which business rules would be violated as well as whether this approach poses any security risks.

Client/Server Models

You often hear of client/server systems falling under one of two models. These are the two-tiered model and three-tiered model, as shown in Figures 29.1 and 29.2, respectively.

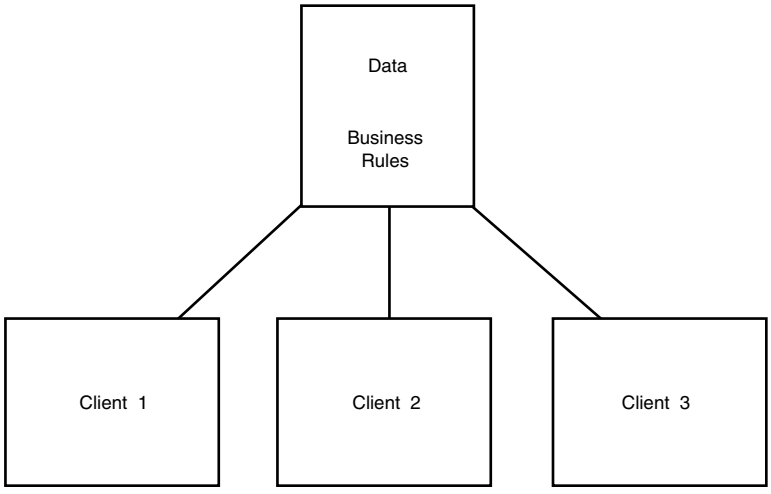

FIGURE 29.1
The two-tiered client/server model.

The Two-Tiered Model

Figure 29.1 illustrates what's referred to as a *two-tiered* client/server model. This model is probably the most common because it follows the same schema as desktop database design. Additionally, many client/server systems being built today have evolved out of existing desktop database applications that stored their data on shared file servers. The migration of systems built around network-shared Paradox or dBASE files up to SQL servers is based on the hope of improved performance, security, and reliability.

Under this model, the data resides on the server, and client applications exist on the client machine. The business logic, or *business rules*, exist on either the client or the server or both.

The Three-Tiered Model

Figure 29.2 shows the *three-tiered* client/server model. Here, the client is the user interface to the data. The remote database server is where the data resides. The client application makes requests to access or modify the data through an applications server or Remote Data Broker. It's typically the Remote Data Broker where the business rules exist.

By distributing the client, server, and business rules on separate machines, designers can more effectively optimize data access and maintain data integrity for other applications in the entire system. Delphi 5 adds powerful capabilities for developing three-tier architectures with the MIDAS technology.

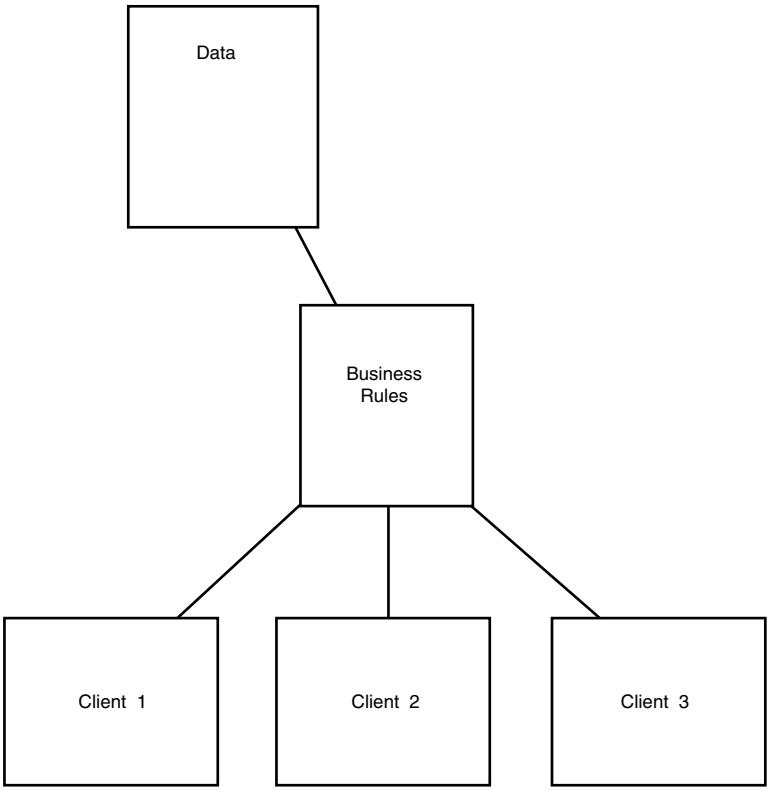

FIGURE 29.2
The three-tiered client/server model.

MIDAS: Multitier Distributed Application Services Suite

Borland's MIDAS technology is included with the Delphi 5 Enterprise version only. This technology is a suite of highly advanced components, servers, and core technologies for your three-tier application development. Chapter 32, "MIDAS Development," discusses this technology in more depth.

Client/Server Versus Desktop Database Development

If you're coming from a background of designing desktop databases, it's important that you understand the differences between desktop database and client/server database development. This next section presents some of the key differences between the two.

Set-Oriented Versus Record-Oriented Data Access

One of the most often misunderstood concepts in client/server development has to do with client/server databases being *set oriented* versus *record oriented*. What this means is that client applications do not work with tables directly as do desktop databases. Instead, client applications work with subsets of the data.

The way this works is that the client application requests rows from the server, which are made up of fields from a table or a combination of several tables. These requests are made using Structured Query Language (SQL).

By using SQL, clients are able to limit the number of records that may be returned from the server. Clients use SQL statements to query the server for a result set, which may consist of a subset of the data on a server. This is an important point to note because when you're accessing desktop databases over a network, the entire table is sent to the calling application across the network. The larger the table, the more this weighs on network traffic. This differs from client/server in that only the requested records are transferred across the network, thus placing fewer requirements on the network.

This difference also affects the navigability of SQL data sets. Concepts such as *first, last*, *next*, and *previous* record are foreign to SQL-based data sets. This is especially true when you think that result sets may consist of rows made up of several tables. Many SQL servers provide *scrollable cursors*, which are navigable pointers on a SQL result set. However, this is not the same as the desktop navigability, which directly navigates through the actual table. You'll see later in the section titled "`TTable` or `TQuery`" how these concepts affect the way you design your client applications with Delphi 5.

Data Security

SQL databases handle security issues differently than do desktop databases. They offer the same password security measures on the overall database access, but they also offer a mechanism to restrict user access to specific database objects such as views, tables, stored procedures, and so on. We'll discuss these objects more later in this chapter. What this means is that user access can be defined on the server based on the user's need to view the data.

Typically, SQL databases allow you to grant or revoke privileges to a user or a group of users. Therefore, it's possible to define a group of users in SQL databases. These privileges may refer to any of the already mentioned database objects.

Record-Locking Methods

Locking is a mechanism used to allow concurrent SQL transactions for many users on the same database. Several locking levels exist, and servers differ as to which level they use.

Table-level locking restricts you from modifying tables that may be involved in an ongoing transaction. Although this method allows for parallel processing, it is slow because users typically need to share the same tables.

An improved locking technique is *page-level* locking. Here, the server locks certain blocks of data on the disk. These are referred to as *pages.* As one transaction is performing an operation on a given page, other transactions are restricted from updating data on that same page. Typically, data is spread over several hundreds of pages, so multiple transactions occurring on the same page are not common.

Some servers offer *record-level* locking, which imposes a lock on a specific row in a database table. However, this results in large overhead in maintaining the locking information.

Desktop databases use what is referred to as *pessimistic* or *deterministic* locking. This means that you're restricted from making changes to table records that are currently being modified by another user. When an attempt to access such a record is made, you'll receive an error message indicating that you cannot access that record until the previous user has freed it.

SQL databases operate on a concept known as *optimistic* locking. With this technique, you aren't restricted from accessing a record that was previously accessed by another user. You can edit and then request the server to save this record. However, before a record is saved, it is compared with the server copy, which may have been updated by another user in the time that you were viewing/editing it on the client end. This will result in an error indicating that the record was modified since you initially received it. As a developer, you must take this into account when designing your client application. Client/server applications must be more reactive to this type of occurrence, which is not the case with their desktop counterparts.

Data Integrity

With SQL databases, you have the opportunity to employ more robust integrity constraints with your server data. Although desktop databases have data integrity constraints built into the database, you must define any business rules in the context of the application's code. In contrast, SQL databases allow you to define these rules on the server end. This gives you the benefit of not only requiring all client applications to use the same set of business rules but also centralizing the maintenance of these rules.

Integrity constraints are defined when you create the tables on the server. We'll show you some samples of this later in the chapter in the section "Creating the Table." Such constraints include validity, uniqueness, and referential constraints.

As stated earlier, integrity constraints can also be defined in the context of the SQL stored procedures. Here, for example, you can check to see if a customer has the proper credit limit before processing an order. You can see how such rules enforce the integrity of the data.

Transaction Orientation

SQL databases are *transaction-oriented*. This means that changes to data aren't made directly to the tables as they are in desktop databases. Instead, the client applications request that the server make these changes, and the server implements this batch of operations in a single transaction.

In order for any changes to the data to be final, the transaction as a whole must be *committed*. If any of the operations within the transaction fails, the entire transaction may be *rolled back* (in other words, aborted).

Transactions preserve the consistency of the data on the server. Let's go back to the inventory example. When an order is made, an `ORDER` table must be updated to reflect the order. Additionally, the `PARTS` table must reflect the reduced number of parts based on the order. If, for some reason, the system fails in between the update to the `ORDERS` table and the update to the `PARTS` table, the data would not correctly reflect the actual number of parts on hand. By encapsulating this entire operation within a transaction, none of the tables affected within the transaction would be updated until the entire transaction is committed.

Transactions can be controlled at the server level or at the client level within your Delphi 5 application. We'll illustrate this later in the chapter in the section "Transaction Control."

NOTE

Some desktop databases, such as Paradox 9, do support transactions.

SQL: Its Role in Client/Server Development

SQL is an industry-standard database-manipulation command set that's used with applications programming environments such as Delphi. SQL is not a language in and of its own. That is, you can't go to the local software store and buy a box of SQL. Instead, SQL is part of the server database.

SQL gained great acceptance as a database query language throughout the '80s and '90s, and today it has become the standard for working with client/server databases across networked environments. Delphi enables you to use SQL through its components. SQL gives you the advantage of viewing your data in the way that only SQL commands will generate, which also gives you much more flexibility than its record-oriented counterpart.

SQL allows you to control the server data by providing the following functionality:

- *Data definition*. SQL lets you define the structures of your tables—the data types of the fields within the tables as well as the referential relationships of certain fields to fields in other tables.
- *Data retrieval*. Client applications use SQL to request from the server whatever data they require. SQL also lets clients define what data to retrieve and how that data is to be retrieved, such as the sorting order, as well as what fields are retrieved.
- *Data integrity*. SQL lets you protect the integrity of the data by using various integrity constraints either defined as part of the table or separately from the table as stored procedures or other database objects.
- *Data processing*. SQL allows clients to update, add, or delete data from the server. This can be as part of a simple SQL statement passed to the server or as a stored procedure that exists on the server.
- *Security*. SQL allows you to protect the data by letting you define user access privileges, views, and restricted access to various database objects.
- *Concurrent access*. SQL manages the concurrent access of data such that users using the system simultaneously don't interfere with each other.

In short, SQL is the primary tool for the development and manipulation of client/server data.

Delphi Client/Server Development

So how does Delphi 5 fit into this client/server environment? Delphi 5 provides you with database object components that encapsulate the functionality of the Borland Database Engine (BDE). This allows you to build database applications without having to know all the functions of the BDE. Additionally, data-aware components communicate with the database-access components. This makes it easy to build user interfaces for database applications. The SQL Links provide native drivers to servers such as Oracle, Sybase, Informix, Microsoft SQL Server, DB2, and InterBase. You can also access data from other databases through ODBC and ADO. In the sections to follow, we'll use both a client/server database—InterBase—and Delphi 5 database components to illustrate various techniques in designing client/server applications.

Delphi 5 includes MIDAS. See the sidebar "MIDAS: Multitier Distributed Application Services Suite" earlier in the chapter or refer to Chapter 34, " Client Tracker: MIDAS Development." Finally, Delphi also gives you the ability to create distributed applications using the Common Object Request Broker Architecture (CORBA). The CORBA specification was adopted by the Object Management Group. This technology gives you the ability to create object-oriented distributed applications. You'll find information on how Delphi 5 handles CORBA in the online help under "Writing CORBA Applications." We simply don't have enough space in this book to provide an adequate discussion of the CORBA technology. This is a topic that merits a book of its own.

The Server: Designing the Back End

When you're designing an application to be built around a client/server environment, quite a bit of planning has to happen before you actually begin coding. Part of this planning process involves defining the business rules for the application. That means deciding which tasks are to be performed on the server and which on the client. Then, you have to decide on table structures and relationships between fields, data types, and user security. In order to accomplish all of this, you should be thoroughly familiar with the database objects on the server end.

For illustration purposes, we'll explain these concepts using InterBase. InterBase is a server database that ships with Delphi. It allows you to create standalone client/server applications that adhere to the ANSI entry-level SQL-92 standard. To use InterBase, you must be familiar with the Windows ISQL program, which ships with Delphi.

NOTE

It's beyond the scope of this book to cover InterBase's implementation of SQL, or any aspect of InterBase for that matter. We're merely using InterBase as a means to discuss client/server application development, which is convenient because the local version of InterBase ships with Delphi 5. Much of what we discuss applies to other implementations of SQL in other server databases, except when it relates to server-specific features.

Database Objects

InterBase uses a Data Definition Language (DDL) to define the various database objects that maintain information about the structure of the database and the data. These objects are also referred to as *metadata*. In the following sections, we describe the various objects that make up the metadata and show examples of how such metadata is defined. Keep in mind that most SQL-based databases consist of similar database objects with which you store information about data.

NOTE

Powerful data-modeling tools such as Erwin, xCase, and RoboCase allow you to graphically design your databases using standard data-modeling methodologies. This is something to consider before you start creating your 200-table system by hand.

Defining Tables

As far as table structure and functionality are concerned, InterBase tables are much like the tables described in Chapter 28, "Writing Desktop Database Applications." That is, they contain an unordered set of rows, each having a certain number of *columns*.

Data Types

Columns can be of any of the available data types, as shown in Table 29.1.

TABLE 29.1 InterBase Data Types

Name	*Size*	*Range/Precision*
`BLOB`	Variable	No limit, 64KB segment size
`CHAR(n)`	*n* characters	1 to 32,767 bytes
`DATE`	64 bits	Jan 1, 100—Dec 11, 5941
`DECIMAL` (precision, scale)	Variable	prec—1 to 15 scale—1 to 15
`DOUBLE PRECISION`	64 bits (platform-dependent)	$1.7x10^{-308}$ to $1.7x10^{308}$
`FLOAT`	32 bits	$3.4x10^{-38}$ to $3.4x10^{38}$
`INTEGER`	32 bits	-2,147,483,648 to 2,147,483,648
`NUMERIC` (precision, scale)	Variable	-32,768 to 32,767
`SMALLINT`	16 bits	1 to 32,767
`VARCHAR(n)`	*n* characters	1 to 32,765

Field types may also be defined with domains in InterBase. We'll discuss this shortly in the section "Using Domains."

Creating the Table

You use the `CREATE TABLE` statement to create the table, its columns, and whatever integrity constraints you want applied to each column. Listing 29.1 shows how you would create an InterBase table.

LISTING 29.1 Table Creation in InterBase

```
/* Domain definitions */
CREATE DOMAIN FIRSTNAME AS VARCHAR(15);
CREATE DOMAIN LASTNAME AS VARCHAR(20);
CREATE DOMAIN DEPTNO AS CHAR(3)
         CHECK (VALUE = '000' OR (VALUE > '0' AND VALUE <= '999')
              OR VALUE IS NULL);
```

continues

LISTING 29.1 Continued

```
CREATE DOMAIN JOBCODE AS VARCHAR(5)
        CHECK (VALUE > '99999');
CREATE DOMAIN JOBGRADE AS SMALLINT
        CHECK (VALUE BETWEEN 0 AND 6);
CREATE DOMAIN SALARY AS NUMERIC(15, 2)
        DEFAULT 0 CHECK (VALUE > 0);

/* Table: EMPLOYEE, Owner: SYSDBA */
CREATE TABLE EMPLOYEE (
        EMP_NO EMPNO NOT NULL,
        FIRST_NAME FIRSTNAME NOT NULL,
        LAST_NAME LASTNAME NOT NULL,
        PHONE_EXT VARCHAR(4),
        HIRE_DATE DATE DEFAULT 'NOW' NOT NULL,
        DEPT_NO DEPTNO NOT NULL,
        JOB_CODE JOBCODE NOT NULL,
        JOB_GRADE JOBGRADE NOT NULL,
        JOB_COUNTRY COUNTRYNAME NOT NULL,
        SALARY SALARY NOT NULL,
        FULL_NAME COMPUTED BY (last_name || ', ' || first_name),
PRIMARY KEY (EMP_NO));
```

The first section of Listing 29.1 shows a series of `CREATE DOMAIN` statements, which we'll explain shortly. The second section of Listing 29.1 creates a table named `EMPLOYEE` with the rows specified. Each row definition is followed by the row type and possibly the `NOT NULL` clause. The `NOT NULL` clause indicates that a value is required for that row. You'll also see that we've specified a primary key on the `EMP_NO` field by using the `PRIMARY KEY` clause. Specifying a primary key not only ensures the uniqueness of the field but also creates an *index* on that field. Indexes speed up data retrieval.

Indexes

Indexes can also be created explicitly by using the `CREATE INDEX` statement. Indexes are based on one or more columns of a table. For example, the following SQL statement would create an index on the last and first names of an employee:

```
CREATE INDEX IDX_EMPNAME ON EMPLOYEE (LAST_NAME, FIRST_NAME);
```

Computed Columns

The `FULL_NAME` field is a computed field. Computed columns are based on a certain expression in the `COMPUTED BY` clause. The example in Listing 29.1 uses the `COMPUTED BY` clause to concatenate the last name and first name, separated by a comma. You can create many variations of computed columns to suit your needs. You should refer to your server documentation to see what capabilities are available for computed columns.

Foreign Keys

You can also specify a foreign key constraint on certain fields. For example, the field DEPT_NO is defined as

```
DEPT_NO DEPTNO NOT NULL
```

The type DEPT NO is defined by its domain. It's okay if you don't understand this for now. Just assume that the field has been given a valid definition such as CHAR(3). To ensure that this field references another field in another table, add the FOREIGN KEY clause to the table definition as shown here, with some of the fields excluded:

```
CREATE TABLE EMPLOYEE (
        EMP_NO EMPNO NOT NULL,
        DEPT_NO DEPTNO NOT NULL
        FIRST_NAME FIRSTNAME NOT NULL,
        LAST_NAME LASTNAME NOT NULL,
PRIMARY KEY (EMP_NO),
FOREIGN KEY (DEPT_NO) REFERENCES DEPARTMENT (DEPT_NO));
```

Here, the FOREIGN KEY clause ensures that the value in the DEPT_NO field of the table EMPLOYEE is the same as a value in the DEPT_NO column in the table DEPARTMENT. Foreign keys also result in an index being created for a column.

Default Values

You can use the DEFAULT clause to specify a default value for a certain field. For example, notice the definition for HIRE_DATE, which uses the DEFAULT clause to specify a default value for this field:

```
HIRE_DATE DATE DEFAULT 'NOW' NOT NULL,
```

Here, the default value to be assigned to this field comes from the result of the NOW function, an InterBase function that returns the current date.

Using Domains

Notice the list of domain definitions that appears before the CREATE TABLE statement. Domains are customized column definitions. By using domains, you can define table columns with complex characteristics that can be used by other tables in the same database. For example, Listing 29.1 shows the domain definition for FIRSTNAME as

```
CREATE DOMAIN FIRSTNAME VARCHAR(15);
```

Any other table that uses FIRSTNAME as one of its field definitions will inherit the same data type, VARCHAR(15). If you find the need to redefine FIRSTNAME later on, any table defining a field of this type inherits the new definition.

You can add constraints to domain definitions as with column definitions. Take, for example, the domain definition for JOBCODE, which ensures that its value is greater than 99999:

```
CREATE DOMAIN JOBCODE AS VARCHAR(5)
        CHECK (VALUE > '99999');
```

You'll also see that the domain JOBGRADE ensures that the value is between 0 and 6:

```
CREATE DOMAIN JOBGRADE AS SMALLINT
        CHECK (VALUE BETWEEN 0 AND 6);
```

The examples provided here are just a mere glimpse of what type of integrity constraints you can place on table definitions. This also varies depending on which type of server you intend to use. It would be to your advantage to be thoroughly familiar with the various techniques provided by your server.

Defining the Business Rules with Views, Stored Procedures, and Triggers

Earlier in the chapter, we talked about business rules—the database logic that defines how data is accessed and processed. Three database objects that allow you to define business rules are *views*, *stored procedures,* and *triggers,* which we discuss in the following sections.

Defining Views

A *view* is a valuable database object that allows you to create a customized result set consisting of clusters of columns from one or more tables in a database. This "virtual table" can have operations performed on it as though it were a real table. This allows you to define the subset of data that a particular user or group of users require in addition to restricting their access to the rest of the data.

To create a view, you would use the CREATE VIEW statement. In InterBase, there are basically three ways to construct a view:

- *A horizontal subset of a single table's rows*. For example, the following view displays all the fields of the EMPLOYEE table with the exception of the SALARY column, which may apply only to management personnel:

  ```
  CREATE VIEW EMPLOYEE_LIST AS
     SELECT EMP_NO, FIRST_NAME, LAST_NAME, PHONE_EXT, FULL_NAME
     FROM EMPLOYEE;
  ```

- *A subset of rows and columns from a single table*. The following example shows a view of employees who are executives based on salaries above $100,000:

  ```
  CREATE VIEW EXECUTIVE_LIST AS
     SELECT EMP_NO, FIRST_NAME, LAST_NAME, PHONE_EXT, FULL_NAME
     FROM EMPLOYEE WHERE SALARY >= 100,000;
  ```

- *A subset of rows and columns from more than one table*. The following view shows a subset of the EMPLOYEE table along with two columns from the JOB table. As far as the client application is concerned, the returned rows/columns belong to a single table:

```
CREATE VIEW ENTRY_LEVEL_EMPL AS
  SELECT JOB_CODE, JOB_TITLE, FIRST_NAME, LAST_NAME.
  FROM JOB, EMPLOYEE
  WHERE JOB.JOB_CODE = EMPLOYEE.JOB_CODE AND SALARY < 15000;
```

Many operations can be applied to views. Some views are read-only, whereas others can be updated. This depends on certain criteria specific to the server you are using.

Defining Stored Procedures

You can think of a *stored procedure* as a standalone routine that's run on the server and invoked from the client applications. Stored procedures are created with the CREATE PROCEDURE statement. There are essentially two types of stored procedures:

- *Select procedures* return a result set of rows consisting of selected columns from one or more tables or a view.
- *Executable procedures* don't return a result set but perform some type of logic on the server side against the server data.

The syntax for defining each type of procedure is the same and consists of a header and a body.

The stored procedure *header* consists of a procedure name, an optional list of parameters, and an optional list of output parameters. The *body* consists of an optional list of local variables and the block of SQL statements that perform the actual logic. This block is enclosed within a BEGIN..END block. The stored procedure can also nest blocks.

A SELECT Stored Procedure

Listing 29.2 illustrates a simple SELECT stored procedure.

LISTING 29.2 A SELECT Stored Procedure

```
CREATE PROCEDURE CUSTOMER_SELECT(
iCOUNTRY        VARCHAR(15)
)
RETURNS(
CUST_NO         INTEGER,
CUSTOMER        VARCHAR(25),
STATE_PROVINCE  VARCHAR(15),
COUNTRY         VARCHAR(15),
POSTAL_CODE     VARCHAR(12)
)
```

continues

LISTING 29.2 Continued

```
AS
BEGIN
 FOR SELECT
   CUST_NO,
   CUSTOMER,
   STATE_PROVINCE,
   COUNTRY,
   POSTAL_CODE
 FROM customer WHERE COUNTRY = :iCOUNTRY
 INTO
    :CUST_NO,
    :CUSTOMER,
    :STATE_PROVINCE,
    :COUNTRY,
    :POSTAL_CODE
 DO
  SUSPEND;
END
 ^
```

This procedure takes an `iCOUNTRY` string as a parameter and returns the specified rows of the `CUSTOMER` table where the country matches that of the `iCOUNTRY` parameter. The code that accomplishes this uses a `FOR SELECT..DO` statement that retrieves multiple rows. This statement functions just like a regular `SELECT` statement except that it retrieves one row at a time and places the specified column values into the variables specified with the `INTO` statement. Therefore, to execute this statement from Windows ISQL, you would enter the following statement:

```
SELECT * FROM CUSTOMER_SELECT("USA");
```

Later, we'll show you how to execute this stored procedure from a Delphi 5 application.

An Executable Stored Procedure

Listing 29.3 illustrates a simple executable stored procedure.

LISTING 29.3 Executable Stored Procedure

```
CREATE PROCEDURE ADD_COUNTRY(
iCOUNTRY        VARCHAR(15),
iCURRENCY       VARCHAR(10)
)
AS
BEGIN
  INSERT INTO COUNTRY(COUNTRY, CURRENCY)
  VALUES (:iCOUNTRY, :iCURRENCY);
```

```
  SUSPEND;
END
 ^
```

This procedure adds a new record to the `COUNTRY` table by issuing an `INSERT` statement with the data passed into the procedure through parameters. This procedure does not return a result set and would be executed by using the `EXECUTE PROCEDURE` statement in Windows ISQL as shown here:

```
EXECUTE PROCEDURE ADD_COUNTRY("Mexico", "Peso");
```

Enforcing Data Integrity Through Stored Procedures

Earlier we stated that stored procedures provide a way of enforcing data integrity on the server, rather than the client. With the stored procedure logic, you can test for integrity rules and raise an error if the client is requesting an illegal operation. As an example, Listing 29.4 performs a "ship order" operation and performs the necessary checks to ensure that the operation is valid. If not, the procedure aborts after raising an exception.

LISTING 29.4 A "Ship Order" Stored Procedure

```
CREATE EXCEPTION ORDER_ALREADY_SHIPPED "Order status is 'shipped.'";
CREATE EXCEPTION CUSTOMER_ON_HOLD "This customer is on hold.";
CREATE EXCEPTION CUSTOMER_CHECK "Overdue balance -- can't ship.";

CREATE PROCEDURE SHIP_ORDER (PO_NUM CHAR(8))
AS

 DECLARE VARIABLE ord_stat CHAR(7);
 DECLARE VARIABLE hold_stat CHAR(1);
 DECLARE VARIABLE cust_no INTEGER;
 DECLARE VARIABLE any_po CHAR(8);
BEGIN
/* First retrieve the order status,
   customer hold information and the customer no
which will be for tests later in the procedure.
    These values are stored in the
local variables defined above. */

 SELECT s.order_status, c.on_hold, c.cust_no
 FROM sales s, customer c
 WHERE po_number = :po_num
 AND s.cust_no = c.cust_no
 INTO :ord_stat, :hold_stat, :cust_no;

 /* Check if the purchase order has been already shipped. If so, raise an
```

continues

LISTING 29.4 Continued

```
    exception and terminate the procedure */

 IF (ord_stat = "shipped") THEN
 BEGIN
  EXCEPTION order_already_shipped;
  SUSPEND;
 END

 /* Check if the Customer is on hold. If so, raise an exception and terminate
    the procedure */

 ELSE IF (hold_stat = "*") THEN
 BEGIN
  EXCEPTION customer_on_hold;
  SUSPEND;
 END

 /* If there is an unpaid balance on orders shipped over 2 months ago,
     put the customer on hold, raise an exception and terminate the procedure
*/

 FOR SELECT po_number
  FROM sales
  WHERE cust_no = :cust_no
  AND order_status = "shipped"
  AND paid = "n"
  AND ship_date < 'NOW' - 60
  INTO :any_po
 DO
 BEGIN
  EXCEPTION customer_check;

  UPDATE customer
  SET on_hold = "*"
  WHERE cust_no = :cust_no;

  SUSPEND;
 END

 /* If we've made it to this point, everything checks out so ship the order.*/
 UPDATE sales
 SET order_status = "shipped", ship_date = 'NOW'
 WHERE po_number = :po_num;

 SUSPEND;
END
 ^
```

You'll notice in Listing 29.4 that the procedure illustrates another feature of InterBase's DDL—*exceptions*. Exceptions in InterBase are much like exceptions in Delphi 5. They are named error messages that are raised from within the stored procedure when an error occurs. When an exception is raised, it returns the error message to the calling application and terminates the execution of the stored procedure. It is possible, however, to handle the exception within the stored procedure and to allow the procedure to continue processing.

Exceptions are created with the `CREATE EXCEPTION` statement, as shown in Listing 29.4. To raise an exception within a stored procedure, you would use the syntax shown in the example and here:

```
EXCEPTION ExceptionName;
```

In Listing 29.4, we define three exceptions that are raised in the stored procedure under various circumstances. The procedure's commentary explains the process that occurs. The main thing to keep in mind is that these checks are being performed within the stored procedure. Therefore, any client application that executes this procedure would have the same integrity constraints enforced.

Defining Triggers

Triggers are basically stored procedures, except they occur upon a certain event and are not invoked directly from the client application or from within another stored procedure. A trigger event occurs during a table *update*, *insert,* or *delete* operation.

Like stored procedures, triggers can make use of exceptions, thus allowing you to perform various data-integrity checks during any of the previously mentioned operations on a particular table. Triggers offer you the following benefits:

- *Data integrity enforcement*. Only valid data can be inserted into a table.
- *Improved maintenance*. Any changes made to the trigger would be reflected by all applications using the table to which the trigger is applied.
- *Automatic tracking of table modifications*. The trigger can log various events that occur on the tables.
- *Automatic notification of table changes through event alerters*.

Triggers consist of a header and a body, just as stored procedures do. The trigger header contains the trigger name, the table name to which the trigger applies, and a statement indicating when a trigger is invoked. The trigger body contains an optional list of local variables and the block of SQL statements that perform the actual logic enclosed between a `BEGIN..END` block, just like a stored procedure.

Triggers are created with the `CREATE TRIGGER` statement. Listing 29.5 illustrates a trigger in InterBase that stores a history of salary changes for employees.

LISTING 29.5 A Trigger Example

```
CREATE TRIGGER SALARY_CHANGE_HISTORY FOR EMPLOYEE
AFTER UPDATE AS
BEGIN
  IF (old.SALARY <> new.SALARY) THEN
    INSERT INTO SALARY_HISTORY (
       EMP_NO,
       CHANGE_DATE,
       UPDATER_ID,
       OLD_SALARY,
       PERCENT_CHANGE)
    VALUES
       old.EMP_NO,
       "now",
       USER,
       old.SALARY,
       (new.SALARY - old.SALARY) * 100 / old.SALARY);
END
```

Let's examine this example more closely. The header contains the following statement:

```
CREATE TRIGGER SALARY_CHANGE_HISTORY FOR EMPLOYEE
AFTER UPDATE AS
```

First, the `CREATE TRIGGER` statement creates a trigger with the name `SALARY_CHANGE_HISTORY`. Then, the statement `FOR EMPLOYEE` specifies to which table the trigger is to be applied; in this case, this is the `EMPLOYEE` table. The `AFTER UPDATE` statement says that the trigger is to be fired after updates to the `EMPLOYEE` table. This statement could have read `BEFORE UPDATE`, which would specify to fire the trigger before changes are made to the table.

Triggers aren't only for updating tables. The following portions of the trigger header can be used in the definition of triggers:

- `AFTER UPDATE`. Fires the trigger after the table is updated
- `AFTER INSERT`. Fires the trigger after a record has been inserted into the table
- `AFTER DELETE`. Fires the trigger after a record is deleted from the table
- `BEFORE UPDATE`. Fires the trigger before updating a record in the table
- `BEFORE INSERT`. Fires the trigger before inserting a new record into the table
- `BEFORE DELETE`. Fires the trigger before deleting a record from the table

Following the `AS` clause in the trigger definition is the trigger body, which consists of SQL statements that form the trigger logic. In the example in Listing 29.5, a comparison is done between the old and new salary. If a difference exists, a record is added to the `SALARY_HISTORY` table indicating the change.

You'll notice that the example makes reference to the identifiers `Old` and `New`. These context variables refer to the current and previous values of a row being updated. `Old` is not used during a record insert, and `New` is not used during a record delete.

You'll see triggers used more extensively in Chapter 32, "Inventory Manager: Client/Server Development," which covers an InterBase client/server application.

Privileges/Access Rights to Database Objects

In client/server databases, users are granted access to or are restricted from accessing data on the server. These *access privileges* can be applied to tables, stored procedures, and views. Privileges are granted by using the `GRANT` statement, which will be illustrated in a moment. First, Table 29.2 illustrates the various SQL access privileges available to InterBase and most SQL servers.

TABLE 29.2 SQL Access Privileges

Privilege	*Access*
`ALL`	The user can select, insert, update, and delete data; see other access rights. `ALL` also grants execute rights on stored procedures.
`SELECT`	The user can read data.
`DELETE`	The user can delete data.
`INSERT`	The user can write new data.
`UPDATE`	The user can edit data.
`EXECUTE`	The user can execute or call a stored procedure.

Granting Access to Tables

To grant user access to a table, you must use the `GRANT` statement, which must include the following information:

- The access privilege
- The table, stored procedure, or view name to which the privilege is applied
- The user's name who is being granted this access

By default, in InterBase only the creator of a table has access to that table and has the ability to grant access to other users. Some examples of granting access follow. You can refer to your InterBase documentation for more information.

The following statement grants `UPDATE` access on the `EMPLOYEE` table to the user with the user name `JOHN`:

```
GRANT UPDATE ON EMPLOYEE TO JOHN;
```

The following statement grants read and edit access on the EMPLOYEE table to the users JOHN and JANE:

```
GRANT SELECT, UPDATE on EMPLOYEE to JOHN, JANE;
```

You can see that you can grant access to a list of users as well. If you want to grant all privileges to a user, use the ALL privilege in your GRANT statement:

```
GRANT ALL ON EMPLOYEE TO JANE;
```

Through the preceding statement, the user JANE will have SELECT, UPDATE, and DELETE access on the table EMPLOYEE.

It's also possible to grant privileges to specific columns in a table, as shown here:

```
GRANT SELECT, UPDATE (CONTACT, PHONE) ON CUSTOMERS TO PUBLIC;
```

This statement grants read and edit access on the fields CONTACT and PHONE in the CUSTOMERS table to all users by using the PUBLIC keyword, which specifies all users.

You must also grant privileges to stored procedures that require access to certain tables. For example, the following example grants read and update access on the customer's table to the stored procedure UPDATE_CUSTOMER:

```
GRANT SELECT, UPDATE ON CUSTOMERS TO PROCEDURE UPDATE_CUSTOMER;
```

The variations on the GRANT statement apply to stored procedures as well.

Granting Access to Views

For the most part, when GRANT is used against a view, SQL treats this just as it would when using GRANT against a table. However, you must be sure that the user to whom you're granting UPDATE, INSERT, and/or DELETE privileges also has the same privileges on the underlying tables to which the view refers. A WITH CHECK OPTION statement used when creating a view ensures that the fields to be edited can be seen through the view before the operation is attempted. It's recommended that modifiable views be created with this option.

Granting Access to Stored Procedures

For users or stored procedures to execute other stored procedures, you must grant them EXECUTE access to the stored procedure to be executed. The following example illustrates how you would grant access to a list of users and stored procedures requiring EXECUTE access to another stored procedure:

```
GRANT EXECUTE ON EDIT_CUSTOMER TO MIKE, KIM, SALLY, PROCEDURE ADD_CUSTOMER;
```

Here, the users MIKE, KIM, and SALLY as well as the stored procedure ADD_CUSTOMER can execute the stored procedure EDIT_CUSTOMER.

Revoking Access to Users

To revoke user access to a table or stored procedure, you must use the `REVOKE` statement, which must include the following items:

- The access privilege to revoke
- The table name/stored procedure to which the revocation is applied
- The user's name whose privilege is being revoked

`REVOKE` looks like the `GRANT` statement syntactically. The following example shows how you would revoke access to a table:

```
REVOKE UPDATE, DELETE ON EMPLOYEE TO JANE, TOM;
```

The Client: Designing the Front End

In the following sections, we'll discuss the Delphi 5 database components and how to use them to access a client/server database. We'll discuss various methods on how to perform common tasks efficiently with these components.

Using the TDatabase Component

The `TDatabase` component gives you more control over your database connections. Here's what it includes:

- Creating a persistent database connection
- Overriding the default server logins
- Creating application-level BDE aliases
- Controlling transactions and specifying transaction isolation levels

Tables 29.3 and 29.4 are brief references to `TDatabase`'s properties and methods. For more detailed descriptions, you'll want to refer to the Delphi online help or documentation. We'll show you how to use some of these properties and methods in this and later chapters.

TABLE 29.3 `TDatabase` Properties

Property	*Purpose*
`AliasName`	An existing BDE alias defined with the BDE Configuration utility. This property cannot be used in conjunction with the `DriverName` property.
`Connected`	A Boolean property to determine whether the `TDatabase` component is linked to a database.

continues

TABLE 29.3 Continued

Property	*Purpose*
`DatabaseName`	Defines an application-specific alias. Other `TDataset` components (`TTable`, `TQuery`, and `TStoredProc`) use this property's value for their `AliasName` property.
`DatasetCount`	The number of `TDataset` components linked to the `TDatabase` component.
`Datasets`	An array referring to all `TDataset` components linked to the `TDatabase` component.
`Directory`	Working directory for a Paradox of dBase database.
`DriverName`	Name of a BDE driver such as Oracle, dBASE, InterBase, and so on. This property cannot be used in conjunction with the `AliasName` property.
`Exclusive`	Give an application sole access to the database.
`Handle`	Used to make direct calls to the Borland Database Engine (BDE) API.
`InTransaction`	Specifies if a transaction is in progress.
`IsSQLBased`	A Boolean property to determine whether the connected database is SQL based. This value is `False` if the `Driver` property holds `STANDARD`.
`KeepConnection`	A Boolean property to determine whether the `TDatabase` maintains a connection to the database when no `TDatasets` are open. This property is used for efficiency reasons because connecting to some SQL servers can take quite a while.
`Locale`	Identifies the language driver used with the `TDatabase` component. This is used primarily for direct BDE calls.
`LoginPrompt`	Determines how the `TDatabase` component handles user logins. If this property is set to `True`, a default login dialog will be displayed. If this property is set to `False`, the login parameters must be provided in code in the `TDataBase.OnLogin` event.
`Name`	The name of the component as referenced by other components.
`Owner`	The owner of the `TDatabase` component.
`Params`	Holds the parameters required to connect to the server database. Default parameters are set using the BDE configuration utility but may be customized here.
`Session`	Points to the session component with which this database component is associated.
`SessionAlias`	Specifies whether or not a database component is using a session alias.
`Tag`	A `longint` property used to store any integer value.
`Temporary`	A Boolean property indicating whether the `TDatabase` component was created as a result of no `TDatabase` component being present when a `TTable`, `TQuery`, or `TStoredProc` was opened.

Property	*Purpose*
`TraceFlags`	Specifies the database operations to track with the SQL Monitor at runtime.
`TransIsolation`	Determines the transaction isolation level for the server.

Table 29.4 lists `TDataBase`'s methods.

TABLE 29.4 `TDataBase` Methods

Method	*Purpose*
`ApplyUpdates`	Posts pending cached updates for specified datasets to the database server.
`Close`	Closes the `TDatabase` connection and all linked `TDataset` components.
`CloseDatasets`	Closes all linked `TDataset` components linked to the `TDatabase` component. This does not necessarily close the `TDatabase` connection.
`Commit`	Commits all changes to the database within a transaction. The transaction must have been established with a call to `StartTransaction`.
`Create`	Allocates memory and creates an instances of a `TDatabase` component.
`Destroy`	Deallocates memory and destroys the `TDatabase` instance.
`Execute`	Executes an SQL statement without the overhead of a TQuery component.
`FlushSchemaCache`	Flushes the cached schema information for a table.
`Free`	Performs the same as `Destroy` except that it first determines whether the `TDatabase` component is set to `nil` before calling destroy.
`Open`	Connects the `TDatabase` component to the server database. Setting the `Connected` property to `True` automatically calls this method.
`RollBack`	Rolls back or cancels a transaction, thus canceling any changes made to the server since the last call to `StartTransaction`.
`StartTransaction`	Begins a transaction with the isolation level specified by the `TransIsolation` property. Modifications made to the server are not committed until a call to the `Commit` method is made. To cancel changes, you must call the `RollBack` method.
`ValidateName`	Raises an exception if a specified database is already open in the active session.

Application-Level Connections

One reason for using a `TDatabase` component with your project is to provide an application-level alias for the entire project. This differs from a BDE-level alias in that the alias name provided by the `TDatabase` component is available only to your project. This application-level alias may be shared among other projects by placing the `TDatabase` component on a sharable

TDataModule. The TDataModule can be made sharable by placing it where other developers can add it to their projects or by placing it into the Object Repository.

You specify the application-level alias by assigning a value to the TDataBase.DatabaseName property. The BDE alias that specifies the server database to which the TDatabase component is connected is specified by the TDatabase.AliasName property.

Security Control

The TDatabase component allows you to control user access to server data in how it handles the login process. During the login process, a user must provide a valid user name and password to gain access to vital data. By default, a standard login dialog is invoked when the user is connected to a server database.

There are several ways you might want to handle logins. One, you can override the login altogether and allow users to gain access to data without having to log in at all. Second, you can provide a different login dialog so that you can perform your own validity checks if necessary before passing the user name and password to the server for normal checks. Finally, you might want to allow users to log off and log in again without shutting down the application. The following sections illustrate all three techniques.

Automatic Login: Preventing the Login Dialog

To prevent the login dialog from displaying when launching an application, you must set the following TDataBase properties:

Property	*Description*
AliasName	Set to an existing BDE alias that was defined with the BDE Administrator. This is the same value typically used as the Alias property value for TTable and TQuery components.
DatabaseName	Set to an application-level alias that will be seen by TDataset descendant components (TTable, TQuery, and TStoredProc) within the current application. These components will use this value as their Alias property value.
LoginPrompt	Set to False. This causes the TDatabase component to look to its Params property to find the user name and password.
Params	Specify the user name and password here. To do this, you must invoke the String List Editor for this property to set the values.

After you've set the TDatabase properties accordingly, you must link all TTable, TQuery, and TStoredProc components to TDatabase by placing the TDatabase.DatabaseName property value as their Alias property value. This value will appear in the drop-down list of aliases when you select the drop-down list in the Object Inspector.

Now, when you set the `TDatabase.Connected` property to `True`, your application will connect to the server without prompting the user for a user name and password because it will use those values defined in the `Params` property. The same will be true when running the application.

You'll find a small example called `NoLogin.dpr` illustrating this on the accompanying CD-ROM.

Providing a Customized Login Dialog

In certain cases, you might want to present your users with a more customized login dialog. For example, you may want to prompt your users for additional information other than just user name and password from the same dialog. Perhaps you just want a more appealing dialog at program startup than that provided by the default login. Whatever the situation, the process is fairly simple.

Basically, you can disable the default login dialog by setting the `TDatabase.LoginPrompt` property to `True`. However, this time you won't provide the user name and password through the `Params` property. Instead, you create an event handler for the `TDatabase.OnLogin` event. This event handler is called whenever the `TDatabase.Connected` property is set to `True` and the `TDatabase.LoginPrompt` property is set to `True`.

The following function instantiates a custom login form and assigns the user's user name and password back to the calling application:

```
function GetLoginParams(ALoginParams: TStrings): word;
var
  LoginForm: TLoginForm;
begin
  LoginForm := TLoginForm.Create(Application);
  try
    Result := LoginForm.ShowModal;
    if Result = mrOK then
    begin
      ALoginParams.Values['USER NAME'] := LoginForm.edtUserName.Text;
      ALoginParams.Values['PASSWORD'] := LoginForm.edtPassWord.Text;
    end;
  finally
    LoginForm.Free;
  end;
end;
```

The `TDataBase.OnLogin` event handler would invoke the preceding procedure as illustrated here (you'll find this sample project on the accompanying CD-ROM as `LOGIN.DPR`):

```
procedure TMainForm.dbMainLogin(Database: TDatabase;
  LoginParams: TStrings);
begin
  GetLoginParams(LoginParams);
end;
```

Logoff During a Current Session

You can also provide functionality for your users to be able to log off and log in again, perhaps as different users, without having to shut down the application. To do this, again you set up the `TDatabase` component so that it does not invoke the default login dialog. Therefore, you must override its `OnLogin` event handler. Also, you must set `TDataBase.LoginPrompt` to `True` so that the event handler will be invoked. The process requires the use of some variables to hold the user name and password as well as a Boolean variable to indicate either a successful or unsuccessful login attempt. Also, you must provide two methods—one to perform the login logic and the other to perform the logoff logic. Listing 29.6 illustrates a project that performs this logic.

LISTING 29.6 Login/Logoff Logic Example

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Grids, DBGrids, BDE, DB, DBTables;

type
  TMainForm = class(TForm)
    dbMain: TDatabase;
    tblEmployee: TTable;
    dsEmployee: TDataSource;
    dgbEmployee: TDBGrid;
    btnLogon: TButton;
    btnLogOff: TButton;
    procedure btnLogonClick(Sender: TObject);
    procedure dbMainLogin(Database: TDatabase; LoginParams: TStrings);
    procedure btnLogOffClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
  public
    TempLoginParams: TStringList;
    LoginSuccess: Boolean;
  end;

var
  MainForm: TMainForm;

implementation
uses LoginFrm;

{$R *.DFM}
```

```
procedure TMainForm.btnLogonClick(Sender: TObject);
begin
  // Get the new login params.
  if GetLoginParams(TempLoginParams) = mrOk then
  begin
    // Disconnect the TDatabase component
    dbMain.Connected := False;
    try
      { Attempt to reconnect the TDatabase component. This will invoke
        the DataBase1Login event handler which will set the LoginParams
        with the current user name and password. }
      dbMain.Connected := True;
      tblEmployee.Active := True;
      LoginSuccess := True;
    except
      on EDBEngineError do
      begin
        //If login failed, specify a failed login and reraise the exception
        LoginSuccess := False;
        Raise;
      end;
    end;
  end;
end;

procedure TMainForm.dbMainLogin(Database: TDatabase;
  LoginParams: TStrings);
begin
  LoginParams.Assign(TempLoginParams);
end;

procedure TMainForm.btnLogOffClick(Sender: TObject);
begin
  { Disconnect the TDatabase component and set the UserName
    and password variables to empty strings }
  dbMain.Connected := False;
  TempLoginParams.Clear;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
  TempLoginParams := TStringList.Create;
end;

procedure TMainForm.FormDestroy(Sender: TObject);
begin
```

continues

LISTING 29.6 Login/Logoff Logic Example

```
  TempLoginParams.Free;
end;

end.
```

In Listing 29.6, you see that the main form has two fields: `TempLoginParams` and `LoginSuccess`. The `TempLoginParams` field holds the user's user name and password. The `btnLogonClick()` method is the logic for the login process, whereas the `btnLogOffClick()` event handler is the logic for the logoff process. The `dbMainLogin()` method is the `OnLogin` event handler for `dbMain`. The code logic is explained in the code commentary. You should also notice that this project uses the same `TLoginForm` used in the previous example. You'll find this example in the project `LogOnOff.dpr` on the accompanying CD-ROM.

Transaction Control

Earlier in this chapter, we spoke of transactions. We mentioned how transactions allow a series of changes to the database to be committed as a whole to ensure database consistency.

Transaction processing can be handled from Delphi 5 client applications by making use of the `TDatabase` properties and methods specific to transactions. The following section explains how to perform transaction processing from within your Delphi 5 application.

Implicit Versus Explicit Transaction Control

Delphi 5 handles transactions either implicitly or explicitly. By default, transactions are handled implicitly.

Implicit transactions are transactions that are started and committed on a row-by-row basis. This means whenever you call a `Post` method or when `Post` is called automatically in VCL code. Because such transactions occur on a row-by-row basis, this increases network traffic, which may lead to efficiency problems.

Explicit transactions are handled in one of two ways. The first method is whenever you call the `StartTransaction()`, `Commit()`, or `RollBack()` method of `TDataBase`. The other method is by using pass-through SQL statements within a `TQuery` component, which we explain momentarily. Explicit transaction control is the recommended approach to use because it provides for less network traffic and safer code.

Handling Transactions

Back in Table 29.4, you saw three methods of `TDatabase` that deal specifically with transactions: `StartTransaction()`, `Commit()`, and `RollBack()`.

`StartTransaction()` begins a transaction using the isolation level specified by the `TDatabase.TransIsolation` property. Any changes made to the server after `StartTransaction()` is called will fall within the current transaction.

If all changes to the server were successful, a call to `TDatabase.Commit()` is made in order to finalize all changes at once. Otherwise, if an error occurs, `TDatabase.RollBack()` is invoked to cancel any changes made.

The typical example of where transaction processing comes in handy has to do with the inventory example. Given an `ORDER` table and an `INVENTORY` table, whenever an order is made, a new record must be added to the `ORDER` table. Likewise, the `INVENTORY` table must be updated to reflect the new item count on hand for the part or parts just ordered. Now suppose that a user enters an order with a system in which transactions were not present. The `ORDER` table gets its new record, but just before the `INVENTORY` table gets updated, a power failure occurs. The database would be in an inconsistent state because the `INVENTORY` table would not accurately reflect the items on hand. Transaction processing would circumvent this problem by ensuring that both table modifications are successful before finalizing any changes to the database. Listing 29.7 illustrates how this might look in Delphi 5 code.

LISTING 29.7 Transaction Processing

```
dbMain.StartTransaction;
  try
     spAddOrder.ParamByName('ORDER_NO').AsInteger := OrderNo;
     { Make other Parameter assignments and then execute the stored
       procedure to add the new order record to the ORDER table.}
     spAddOrder.ExecProc;
     { Iterate through all the parts ordered and update the
       INVENTORY table to reflect the # of parts on hand }
     for i := 0 to PartList.Count - 1 do
     begin
       spReduceParts.ParamByName('PART_NO').AsInteger :=
         PartRec(PartList.Objects[i]).PartNo;
       spReduceParts.ParamByName('NUM_SOLD').AsInteger :=
         PartRec(PartList.Objects[i]).NumSold;
       spReduceParts.ExecProc;
     end;
     // Commit the changes to both the ORDER and INVENTORY tables.
     dbMain.Commit;
  except
    // If we get here, an error occurred. Cancel all changes.
    dbMain.RollBack;
    raise;
  end;
```

This code is a simplistic example of how to use transaction processing to ensure database consistency. It uses two stored procedures—one to add the new order record and the other to update the `INVENTORY` table with the new data. Keep in mind that this is just a code snippet to illustrate transaction processing with Delphi. This logic could probably be handled better on the server side.

In some cases, the type of transaction processing that must happen might depend on server-specific features. Given this situation, you would have to use a `TQuery` component to pass the server-specific SQL code, which requires that you set the SQL pass-through mode accordingly.

SQL Pass-through Mode

The SQL pass-through mode specifies how Delphi 5 database applications and the Borland Database Engine (BDE) share connections to database servers. The BDE connections are those used in Delphi methods that make BDI API calls. The pass-through mode is set in the BDE Configuration utility. The three settings for the pass-through mode are as follows:

Setting	*Description*
`SHARED AUTOCOMMIT`	Transactions are handled on a row-by-row basis. This method is more closely related to that of desktop databases. In the client/server world, this causes heavy network traffic and is not the recommended approach. However, this is the default setting for Delphi 5 applications.
`SHARED NOAUTOCOMMIT`	Delphi 5 applications must explicitly start, commit, and cancel transactions using the `TDatabase.StartTransaction()`, `Commit()`, and `RollBack()` methods.
`NOT SHARED`	The BDE and `TQuery` components issuing pass-through SQL statements do not share the same connections. This means that the SQL code is not restricted to BDE capabilities and may consist of server-specific features.

If you're not using pass-through SQL but want more control over your transaction processing, set the pass-through mode to `SHARED NOAUTOCOMMIT` and handle the transaction processing yourself. In most cases, this should suit your needs. Just keep in mind that in multiuser environments where the same rows get updated often, conflicts may occur.

Isolation Levels

Isolation levels determine how transactions see data that's being accessed from other transactions. The `TDatabase.TransIsolation` property determines what isolation level a particular transaction will use. There are three isolation levels to which you can assign the `TransIsolation` property:

Isolation Level	*Description*
`tiDirtyRead`	The lowest isolation level. Transactions using this isolation level can read uncommitted changes from other transactions.
`tiReadCommitted`	The default isolation level. Transactions using this isolation level can read only committed changes by other transactions.
`tiRepeatableRead`	This is the highest isolation level. Transactions using this isolation level cannot read changes to previously read data made by other transactions.

The support for the isolation levels listed here may vary on different servers. Delphi 5 will always use the next highest isolation level if a specific isolation level is not supported.

TTable or TQuery

A common misunderstanding is the idea that developing front-end client applications is the same as or similar to developing desktop database applications.

Where you'll see this frame of thinking manifest itself is in how or when one uses `TTable` versus `TQuery` components for database access. In the following paragraphs, we'll discuss some of the merits and faults of using a `TTable` component, when it should be used, and when it should not be used. You'll also see why you're most often better off using a `TQuery` component.

Can TTable Components Do SQL?

`TTable` components are great for accessing data in a desktop environment. They are designed to perform the tasks that desktop databases require such as manipulation of the entire table, navigation forward and backward through a table, and even going to a specific record in the table. These concepts, however, are foreign to SQL database servers. Relational databases are designed to be accessed in sets of data. SQL databases do not know the concepts of "next," "previous," and "last" record—something that `TTable` is good at. Although some SQL databases provide "scrollable cursors," this is not a standard and typically applies only to the result set. Additionally, some servers don't provide bidirectional scrolling.

The key point to make when comparing `TTable` components against SQL databases is that, ultimately, the commands issued through `TTable` must be converted to SQL code that the SQL database can understand. Not only does this limit how you can access the server, but it also weighs heavily on efficiency.

To demonstrate the inherent weakness of using `TTable` to access large datasets, consider the process of opening a `TTable` just to retrieve a few records. The time it takes for a `TTable` to open a SQL table is directly proportional to the number of fields and the amount of metadata

(index definitions and so on) attached to the SQL table. When you issue a command such as the following against a SQL table, the BDE sends a series of SQL commands to the server to first retrieve information about the table's columns, indexes, and so on:

```
Table1.Open;
```

Then it issues a `SELECT` statement to build a result set consisting of all the columns and rows from the table. This is where the time it takes to open a table might also be proportional to the size of the SQL table (the number of rows). Even though only the amount of rows necessary to populate the data-aware components are returned to the client, an entire result set is being built in response to the query. This process occurs whenever the `TTable` is opened. On extremely large tables, typical with client/server databases, this single operation can take up to 20 seconds. Keep in mind that some SQL servers such as Sybase and Microsoft SQL don't allow a client to abort the retrieval of a result set. This is where the table's size affects the select duration. Oracle, InterBase, and Informix all enable you to abort a result set without this considerable penalty.

Despite the disadvantages to using `TTables` with a client application, they are typically fine for accessing small tables on the server. You have to test your applications to determine whether the performance hit is unacceptable.

NOTE

MIDAS handles the returning of data packets a bit differently. You'll want to read about this in Chapter 34, "Client Tracker: MIDAS Development."

Issuing FindKey and FindNearest Against SQL Databases

Although `TTable` is capable of looking up records using the `FindKey()` method, it has its limitations when using this against a SQL database. First, `TTable` can only use `FindKey` against an indexed field or fields if you're performing a search based on values from multiple fields. `TQuery` is not faced with this limitation because you perform the record search through SQL. It's true that `TTable.FindKey` results in a `SELECT` statement against the server table. However, the result set will consist of all fields of the table even though you may have only selected certain fields from the `TTable` component's Fields Editor.

To achieve the functionality of `FindNearest()` with SQL code is not as straightforward as using a `TTable`, yet it's not impossible. The following SQL statement almost accomplishes the `TTable.FindNearest()` functionality:

```
SELECT * FROM EMPLOYEES
  WHERE NAME >= "CL"
  ORDER BY NOMENCLATURE
```

Here, the result set returns the record either *at* the position searched for or directly after where it *should be*. The problem here is that this result set returns *all* the records after the position searched on. To be more accurate so that the result set will consist of only one record, you can do the following:

```
SELECT * FROM EMPLOYEES
   WHERE NAME = (SELECT MIN(NAME) FROM EMPLOYEES
   WHERE NAME >= "CL")
```

Here, you use a nested `SELECT`. In a nested `SELECT` statement, the inner statement returns its result set to the outer `SELECT`. The outer `SELECT` then uses this result set to process its statement. In the inner query in this example, you use the SQL aggregate function `MIN()` to return the lowest value in the column `NAME` on the table `EMPLOYEES`. This single-row single-column result set is then used in the outer query to retrieve the remaining rows.

The point is that you give yourself much more flexibility and efficiency by maximizing SQL capabilities and using the `TQuery` component. By using `TTable`, you only limit what you're able to do against the server data.

Using the TQuery Component

In the previous chapter, you were introduced to the `TQuery` component and shown how you can use it to retrieve result sets of rows in tables. We're going to get a bit more into detail on `TQuery` in the following sections. We'll illustrate how to create dynamic SQL statements at runtime, how to pass parameters to queries, and how to improve `TQuery` performance by setting certain property values.

There are basically two types of queries for which you'll use `TQuery`: those that return result sets and those that don't. For queries returning a result set, you use the `TQuery.Open()` method. The `TQuery.ExecSQL()` method is used when a result set is not returned.

Dynamic SQL

Dynamic SQL means that you can modify your SQL statements at runtime based on various conditions. When you invoke the String List Editor for the `TQuery.SQL` property and enter a statement such as the following, you're entering a static SQL statement:

```
SELECT * FROM EMPLOYEE WHERE COUNTRY = "USA"
```

This statement won't vary unless you completely replace it at runtime.

To make this statement dynamic, you would enter the following into the `SQL` property:

```
SELECT * FROM CUSTOMER WHERE COUNTRY = :iCOUNTRY;
```

In this statement, instead of hard-coding the value on which to search, we've provided a placeholder, a *parameter* whose value can be specified later. This variable is named `iCountry` and

follows the colon in the `SELECT` statement. Its name was chosen at random. Now, you can search on any country by providing the country string to search on.

There are several ways to provide values for a parameterized query. One way is to use the property editor for the `TQuery.Params` property. Another is to provide that value at runtime. You can also provide the value from another dataset through a `TDataSource` component.

Providing TQuery Parameters Through the Params Property Editor

When you invoke the `TQuery.Params` property editor, the Parameter Name list displays the parameters for a given query. For each parameter listed, you must select a type from the Data Type drop-down combo. The value field is where you can specify an initial value for the parameter if you like. You can also select the NULL check box to set the parameter's value to null. When you select OK, the query will prepare its parameters, which binds them to their types (see the sidebar titled "Preparing Queries"). When you invoke `TQuery.Open()`, a result set will be returned to the `TQuery`.

Preparing Queries

When a SQL statement gets sent to the server, the server must parse, validate, compile, and execute the statement. This happens every time you send a SQL statement to the server. You can improve performance by allowing the server to perform the preliminary steps of parsing, validating, and compiling by "preparing" the SQL statement before having the server execute it. This is especially advantageous when using a query repetitively in a loop, by calling `TQuery.Prepare()` before entering the loop as shown in the following code:

```
Query1.Prepare; // First prepare the query.
try
  { Enter a loop to execute a query numerous times }
  for i := 1 to 100 do begin
    { provide the parameters for the query }
    Query1.ParamByName('SomeParam').AsInteger := i;
    Query1.ParamByName('SomeOtherParam').AsString := SomeString;
    Query1.Open;  // Open the query.
    try
      { Use the result set of Query1 here. }
    finally
      Query1.Close; // Close the query.
    end;
  end;
 finally
  Query1.Unprepare; // Call unprepare to free up resources
end;
```

Prepare() only needs to be called once before its repetitive use. You can also change the values of the query parameters after the first call to Prepare() without having to call Prepare() again. However, if you change the SQL statement itself, you must call Prepare() again before reusing it. A call to Prepare() must be matched with a call to TQuery.UnPrepare() to release the resources allocated by Prepare().

Queries get prepared when you select the OK button on the Params property editor, or when you call the TQuery.Prepare() method, as shown in the preceding code. It's also recommended that you call Prepare() once in the form's OnCreate event handler and UnPrepare() in the form's OnDestroy event handler for those queries whose SQL statements won't change. It's not necessary to prepare your SQL queries, but it's certainly beneficial to do so.

Providing TQuery Parameters Using the Params Property

The TQuery component has a zero-based array of TParam objects, each representing parameters of the SQL statement in the TQuery.SQL property. For example, take a look at the following SQL statement:

```
INSERT INTO COUNTRY (
  NAME,
  CAPITAL,
  POPULATION)
VALUES(
  :NAME,
  :CAPITAL,
  :POPULATION)
```

To use the Params property to provide values for the parameters :Name, :CAPITAL, and :POPULATION, you would issue the following statement:

```
with Query1 do begin
  Params[0].AsString := 'Peru';
  Params[1].AsString := 'Lima"
  Params[2].AsInteger := 22,000,000;
end;
```

The values provided would be bound to the parameters in the SQL statement. Keep in mind that the order of the parameters in the SQL statement dictates their position in the Params property.

Providing TQuery Parameters Using the ParamByName Method

In addition to the Params property, the TQuery component has the ParamByName() method. The ParamByName() method enables you to assign values to the SQL parameters by their name rather than by their position in the SQL statement. This enhances code readability but isn't as efficient as the positional method because Delphi must resolve the parameters being references.

To use the ParamByName() method to provide value for the preceding INSERT query, you would use the following code:

```
with Query1 do begin
  ParamByName('COUNTRY').AsString := 'Peru';
  ParamByName('CAPITAL').AsString := 'Lima';
  ParamByName('POPULATION').AsInteger := 22,000,000;
end;
```

You should see that this code is a bit clearer as to which parameters you're providing values.

Providing TQuery Parameters Using Another Dataset

The parameters provided to a TQuery component can also be gotten from another TDataset such as TQuery or TTable. This creates a master-detail relationship between the two datasets. To do this, you must link a TDataSource component to the master dataset. The name of this TDataSource is assigned to the DataSource property of the detail TQuery component. When the query is executed, Delphi checks to see whether any value is assigned to the TQuery.DataSource property. If so, it will look for column names of the DataSource that match parameter names in the SQL statement and will then bind them.

As an example, consider the following SQL statement:

```
SELECT * FROM SALARY_HISTORY
  WHERE EMP_NO = :EMP_NO
```

Here, you need a value for the parameter named EMP_NO. First, you assign the TDataSource that refers to the master TTable component to the TQuery's DataSource property. Delphi will then search for a field named EMP_NO in the table to which the TTable refers and will bind the value of that column to the TQuery's parameter for the current row. This is illustrated in the example found in the project LnkQuery.dpr on the accompanying CD-ROM.

Using the Format Function to Design Dynamic SQL Statements

Now that we've shown you how to use parameterized queries, it might seem reasonable that either of the following SQL statements would be valid:

```
SELECT * FROM PART ORDER BY :ORDERVAL;
SELECT * FROM :TABLENAME
```

Unfortunately, you cannot replace certain words in a SQL statement such as column names and table names. SQL servers just don't support this capability. So how do you go about putting this type of flexibility into your dynamic SQL statements? You do this by constructing your SQL statements at design time by using the Format() function.

If you have any experience programming in C or C++, you'll find that the Format() function works much like C's printf() function. See the sidebar on the Format() function.

Using the Format() Function

Use the `Format()` function to customize strings that vary depending on values provided by *format specifiers*. Format specifiers are placeholders where strings of a specified type will be inserted into a given string. These specifiers consist of a percent symbol (`%`) and a *type specifier*. The following list illustrates some type specifiers:

Specifier	*Description*
`c`	Specifies a char type
`d`	Specifies an integer type
`f`	Specifies a float type
`p`	Specifies a pointer type
`s`	Specifies a string type

For example, in the string `"My name is %s and I'm %d years old."`, you see two format specifiers. The `%s` specifier indicates that a string is to be inserted in its place. The `%d` specifier indicates that an integer is to be inserted in its place. To construct the string, here's how to use the `Format()` function:

```
S := Format('My name is %s and I'm %d years old.", ['Xavier', 32]);
```

The `Format()` function takes the source string and an open array of arguments to replace the format specifiers. It returns the resulting string. You'll find detailed information on the `Format()` function in Delphi 5's online help.

Therefore, to construct SQL statements with the flexibility to modify field names or table names, you can use the `Format()` function as illustrated in the following code examples.

Listing 29.8 illustrates how you would use the `Format()` function to allow the user to pick the fields by which to sort the result set of a query. The list of fields exists in a list box, and the code is actually the `OnClick` event of that list box. You'll find this demo in the project `OrderBy.dpr` on the accompanying CD-ROM.

LISTING 29.8 Using `Format()` to Specify Sorting Column

```
procedure TMainForm.lbFieldsClick(Sender: TObject);
{ Define a constant string from which the SQL string will be built }
const
   SQLString = 'SELECT * FROM PARTS ORDER BY %s';
begin
  with qryParts do
  begin
    Close;     // Make sure the query is closed.
```

continues

LISTING 29.8 Continued

```
    SQL.Clear; // Clear any previous SQL statement.
    { Now add the new SQL statement constructed with the format
      function }
    SQL.Add(Format(SQLString, [lbFields.Items[lbFields.ItemIndex]]));
    Open;  { Now open Query1 with the new statement }
  end;
end;
```

To populate the list box in Listing 29.8 with the field names in the parts table, we used the following code in the form's `OnCreate` event handler:

```
  tblParts.Open;
  try
    tblParts.GetFieldNames(lbFields.Items);
  finally
    tblParts.Close;
  end;
```

`tblParts` is linked to the `PARTS.DB` table.

The next example in Listing 29.9 illustrates how to pick a table on which to perform a `SELECT` statement. The code is practically the same as that presented in Listing 29.8, except that the format string is different and the form's `OnCreate` event handler retrieves a list of table names in the given session rather than a list of fields for a single table.

First, a list of table names is obtained:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
{  First, get a list of table names for the user to select }
  Session.GetTableNames(dbMain.DatabaseName, '', False, False, lbTables.Items);
end;
```

Then, the `lbTables.OnClick` event handler is used to select the table on which to perform a `SELECT` query, as shown in Listing 29.9.

LISTING 29.9 Using `Format()` to Specify a Table to Select

```
procedure TMainForm.lbTablesClick(Sender: TObject);
{ Define a constant string from which the SQL string will be built }
const
   SQLString = 'SELECT * FROM %s';
begin
  with qryMain do
  begin
    Close;     // Make sure the query is closed.
```

```
    SQL.Clear; // Clear any previous SQL statement.
    { Now add the new SQL statement constructed with the format
      function }
    SQL.Add(Format(SQLString, [lbTables.Items[lbTables.ItemIndex]]));
    Open;  { Now open Query1 with the new statement }
  end;
end;
```

This demo is provided in the project `SelTable.dpr` on the accompanying CD-ROM.

Retrieving the Result Set Values of a Query Through TQuery

When a query operation returns a result set, you can access the values of the columns in that result set by using the `TQuery` component as though it were an array whose field names are indexes into this array. For example, suppose you have a `TQuery` whose `SQL` property contains the following SQL statement:

```
SELECT * FROM CUSTOMER
```

You would retrieve the values of the columns as shown in Listing 29.10, which shows the code for the project `ResltSet.dpr` on the accompanying CD-ROM.

LISTING 29.10 Retrieving the Fields of a `TQuery` Result Set

```
procedure TMainForm.dsCustomerDataChange(Sender: TObject; Field: TField);
begin
  with lbCustomer.Items do
  begin
    Clear;
    Add(VarToStr(qryCustomer['CustNo']));
    Add(VarToStr(qryCustomer['Company']));
    Add(VarToStr(qryCustomer['Addr1']));
    Add(VarToStr(qryCustomer['City']));
    Add(VarToStr(qryCustomer['State']));
    Add(VarToStr(qryCustomer['Zip']));
    Add(VarToStr(qryCustomer['Country']));
    Add(VarToStr(qryCustomer['Phone']));
    Add(VarToStr(qryCustomer['Contact']));
  end;
end;
```

In the preceding code, you use the default dataset method, `FieldValues()`, to access the field values of `qryCustomer`. Because `FieldValues()` is the default dataset method, it's not necessary to specify the method name explicitly, as shown here:

```
Add(VarToStr(qryCustomer.FieldValues['Contact']));
```

CAUTION

The function `FieldValues()` returns a variant field type. If a field were to contain a null value, an attempt to get the field's value with `FieldValue()` would result in an `EVariantError` exception. Therefore, Delphi provides the `VarToStr()` function, which converts null string values to an empty string. Equivalent functions for other data types are not provided. However, you can construct your own as shown here for integer types:

```
function VarToInt(const V: Variant): Integer;
begin
  if TVarData(V).VType <> varNull then
    Result := V
  else
    Result := 0;
end;
```

Be careful, however, when you resave the data. A null value in a SQL database is a valid value. If you were to replace that value with an empty string, which is not the same as null, you could destroy the integrity of the data. You'll have to come up with a runtime solution to this, such as testing for `NULL` and storing some predefined string to represent the null value.

You can also retrieve the field values from a `TQuery` using the `TQuery.Fields` property. The `Fields` property is used in the same way as the `TQuery.Params` property, except it refers to the columns in the result set. Similarly, `TQuery` has the `FieldByName()` method, which functions like the `ParamByName()` method.

The UniDirectional Property

To optimize access to a database, the `TQuery` component has the `UniDirectional` property. This applies to databases that support *bidirectional cursors*. Bidirectional cursors enable you to move forward and backward through the query's result set. By default, this property is `False`. Therefore, when you have components such as the `TDBGrid` component linked to a database that does not support bidirectional movement, Delphi emulates this movement by buffering records on the client side. This can take up a lot of resources on the client end rather quickly. Therefore, if you plan to only move forward through a result set, or if you plan to go through the result set only once, set `UniDirectional` to `True`.

Live Result Sets

By default, `TQuery` returns read-only result sets. You can specify for `TQuery` to return a modifiable result set by changing the `TQuery.RequestLive` property to `True`. However, certain restrictions apply to doing this, as shown in the following lists.

For queries returning result sets from dBASE or Paradox tables, these restrictions apply:

- Uses local SQL Syntax (information provided in online help).
- Uses only a single table.
- SQL statement does not use an `ORDER BY` clause.
- SQL statement does not use aggregate functions such as `SUM` and `AVG`.
- SQL statement does not use calculated fields.
- Comparisons in the `WHERE` clause may consist only of column names to scalar types.

For queries using pass-through SQL from a server table, these restrictions apply:

- Uses a single table.
- SQL statement does not use an `ORDER BY` clause.
- SQL statement does not use aggregate functions such as `SUM` and `AVG`.

To determine whether a query can be modified, you can check the `TQuery.CanModify` property.

Cached Updates

`TDataSets` contain a `CachedUpdate` property, which allows you to turn any query or stored procedure into an updateable view. This means the changes to the data set are written to a temporary buffer on the client instead of these changes being written to the server. These changes can then be sent to the server by calling the `ApplyUpdates()` method for the `TQuery` or `TStoredProc` component. Cached updates allow optimization of the updates and remove much of the lock contention on the server. You might refer to Chapter 13 of "Delphi 5 Database Application Developer's Guide," of the Delphi 5 documentation which is dedicated to working with cached updates.

Executing Stored Procedures

Delphi's `TStoredProc` and `TQuery` components are both capable of executing stored procedures on the server. The following sections explain how to use both components to perform stored procedure execution.

Using the TStoredProc Component

The `TStoredProc` component enables you to execute stored procedures on the server. Depending on the server, it can return either a singleton or multiple result set. `TStoredProc` may also execute stored procedures that return no data at all. To execute server stored procedures, the following `TStoredProc` properties must be set accordingly:

Property	*Description*
`DataBaseName`	The name of the database that contains the stored procedure. This is usually the `DataBaseName` property for the `TDatabase` component referring to this server database.
`StoredProcName`	The name of the stored procedure to execute.
`Params`	This contains the input and output parameters defined by the stored procedure. The order is also based on the definition of the stored procedure on the server.

TStoredProc Input and Output Parameters

You provide input and output parameters through the `TStoredProc.Params` property. Like `TQuery`, the parameters must be *prepared* with default data types. This can be done either at design time through the Parameters Editor or at runtime, as will be illustrated.

To prepare parameters using the Parameters Editor, you right-click the `TStoredProc` component to invoke the Parameters Editor.

The Parameters Name list box shows a list of the input and output parameters for the stored procedure. Note that you must have already selected a `StoredProcName` from the server for any parameter to display. For each parameter, you specify a data type in the Data Type drop-down combo box. You can also specify an initial value or a null value, as with the `TQuery` component. When you select the OK button, the parameters will be prepared.

You can also prepare the `TStoredProc`'s parameters at runtime by executing the `TStoredProc.Prepare()` method. This function is just like the `Prepare()` method for the `TQuery` component discussed earlier.

Executing Non–Result Set Stored Procedures

To understand executing a stored procedure that does not return a result set, see Listing 29.11, which shows an InterBase stored procedure that adds a record to a `COUNTRY` table.

LISTING 29.11 Insert `COUNTRY` Stored Procedure in InterBase

```
CREATE PROCEDURE ADD_COUNTRY(
iCOUNTRY        VARCHAR(15),
iCURRENCY       VARCHAR(10)
)
AS
BEGIN
  INSERT INTO COUNTRY(COUNTRY, CURRENCY)
  VALUES (:iCOUNTRY, :iCURRENCY);
  SUSPEND;
END
 ^
```

To execute this stored procedure from Delphi, you would first set up the `TStoredProc` component with the appropriate values for the properties specified earlier. This includes specifying the parameter types from the Parameters Editor. The Delphi code to run this stored procedure is presented in Listing 29.12.

LISTING 29.12 Executing a Stored Procedure Through `TStoredProc`

```
with spAddCountry do
  begin
    ParamByName('iCOUNTRY').AsString := edtCountry.Text;
    ParamByName('iCURRENCY').AsString := edtCurrency.Text;
    ExecProc;
    edtCountry.Text := '';
    edtCurrency.Text := '';
    tblCountries.Refresh;
  end;
```

Here, you first assign the values from two `TEdits` to the `TStoredProc` parameters through the `ParamByName()` method. Then you call the `TStoredProc.ExecProc()` function, which executes the stored procedure. You'll find an example that illustrates this code in the project `AddCntry.dpr`.

NOTE

To run the `AddCntry.dpr` project, you must use the `BDEADMIN.EXE` utility to set up a new alias named "DB." This alias must point to the file `\CODE\DATA\DDGIB.GDB`, which can be found on the CD-ROM accompanying this book. Refer to the documentation for the BDE Administrator utility for further information.

Getting a Stored Procedure Result Set from TQuery

It's also possible to execute a stored procedure using a pass-through SQL statement with a `TQuery` component. This is necessary in some cases, as with InterBase, which doesn't support stored procedures that must be called with a `SELECT` statement. For example, a stored procedure that returns result sets can be called just as though it were a table. Take a look at Listing 29.13, which is an InterBase stored procedure that returns a list of employees from an `EMPLOYEE` table belonging to a particular department. The department is specified by the input parameter `iDEPT_NO`.

LISTING 29.13 GET_EMPLOYEES_BY_DEPT Stored Procedure

```
CREATE PROCEDURE GET_EMPLOYEES_IN_DEPT (
iDEPT_NO           CHAR(3))
RETURNS(
EMP_NO           SMALLINT,
FIRST_NAME       VARCHAR(15),
LAST_NAME        VARCHAR(20),
DEPT_NO          CHAR(3),
HIRE_DATE        DATE)
AS
BEGIN
  FOR SELECT
    EMP_NO,
    FIRST_NAME,
    LAST_NAME,
    DEPT_NO,
    HIRE_DATE
  FROM EMPLOYEE
  WHERE DEPT_NO = :iDEPT_NO
  INTO
     :EMP_NO,
     :FIRST_NAME,
     :LAST_NAME,
     :DEPT_NO,
     :HIRE_DATE
  DO
    SUSPEND;
END ^
```

To execute this stored procedure from within Delphi 5, you need to use a `TQuery` component with the following `SQL` property:

```
SELECT * FROM GET_EMPLOYEES_IN_DEPT(
:iDEPT_NO)
```

Notice that this statement uses the `SELECT` statement as though the procedure were a table. The difference, as you can see, is that you must also provide the input parameter `iDEPT_NO`.

We've created a sample project, `Emp_Dept.dpr`, that illustrates executing the preceding stored procedure.

`qryGetEmployees` is the `TQuery` component that executes the stored procedure shown in Listing 29.13. It gets its parameter from `qryDepartment`, which performs a simple `SELECT` statement on the `DEPARTMENT` table in the database. `qryGetEmployees` is linked to `dbgEmployees`, which shows a scrollable list of departments. When the user scrolls through

dbgDepartment, this invokes dsDepartment's OnDataChange event handler. We should mention that dsDepartment is linked to qryDepartment. This event handler executes the code shown in Listing 29.14, which sets the parameter for qryGetEmployees and retrieves its output result set.

LISTING 29.14 DataSource1's OnChange Event Handler

```
procedure TMainForm.dsDepartmentDataChange(Sender: TObject; Field: TField);
begin
  with qryGetEmployees do
  begin
   Close;
   ParamByName('iDEPT_NO').AsString := qryDepartment['DEPT_NO'];
   Open;
  end;
end;
```

So why would you want to retrieve this information through a stored procedure rather than a simple statement on a table? Consider that there may be several people at different levels within a department who need access to the information provided. If these people had direct access to the table, they would be able to see sensitive information such as an employee's salary. By restricting access to a table but providing the "need to know" information through stored procedures and views, you not only establish good security measures but also create a more maintainable set of business rules for the database.

Summary

This chapter presented you with quite a bit of information about client/server development. We first discussed the elements that make up a client/server system. We compared client/server development to traditional desktop database development methodologies. We also introduced you to various techniques using Delphi 5 and InterBase that should set you well on your way to developing client/server projects.