

Virtual Database Tutorial

Content

1 Virtual Database 1

1.1 Introduction 1

1.1.1 Overview 1

1.1.2 Example 1

1.2 Connecting 3

1.2.1 Configuring Your Application to Use VDB 3

1.2.2 Profile Strings 3

1.2.3 Creating a Connection 5

1.3 VDB Shortcuts 6

1.3.1 NewQuery 6

1.3.2 RunQuery 6

1.3.3 ExecSQL 6

1.3.4 QueryValue and QueryValueDef 6

1.3.5 Parameterized SQL Shortcuts 7

1.3.6 SQL Formatting Shortcuts 9

1.3.7 Transaction Shortcuts 10

1.4 Vendor Neutral SQL 11

1.4.1 Function Escapes 11

1.4.2 Dialects 14

1.5 Advanced Configuration 14

1.5.1 Connection Pooling 14

1.5.2 Automatic, Adaptive Query Preparing and Caching 15

1.5.3 Linked Profile Strings 17

1.6 N-Tier Development 17

1.6.1 VDB Remote Datasets 17

1.6.2 Connection Pooling in Distributed Applications 18

2 Index 21

1 Virtual Database

1.1 Introduction

Welcome to Virtual Database (VDB) from Dimeric Software! You can contact us and find out more about VDB and our other products at <http://www.Dimeric.com/>

Copyright (c) 2002 by Dimeric, L.L.C. All rights reserved.

1.1.1 Overview

VDB is a database library for Delphi. VDB stands for virtual database, because it abstracts Delphi's numerous database libraries. VDB has several advantages:

- **Database Library Portability.** You can easily switch between database libraries without even recompiling the program. This gives your applications the freedom to work with the Borland Database Engine, Microsoft's ADO, InterBase Express, DB/Express, or other third-party database libraries for Delphi. (See Connecting)
- **Simplified Coding.** The VDB API is modeled primarily using interfaces (IDatabase, IDataset, IQuery, etc.). This means that VDB objects are managed, and resources are released automatically. In addition, while providing the familiar properties, methods and events of the built-in objects (TDatabase, TQuery, etc.) VDB provides additional simplified methods for performing common tasks. (See VDB Shortcuts)
- **SQL Dialect Abstraction.** VDB provides abstraction for the underlying database SQL dialect. VDB provides an escape syntax so that you can write the same SQL regardless of the database you are connecting to. This escape syntax is automatically translated into appropriate code for the database's SQL dialect. For example, the macro for the database's current date and time would be CURRENT for the Informix dialect and getdate() for the SQL Server dialect. In VDB you would use the [now()] function. If you were connecting to Informix, this would be translated into CURRENT or getdate() for SQL Server. This makes code written to VDB more portable across SQL dialects. (See Vendor Neutral SQL)
- **Connection Pooling.** If you set a few properties in a configuration string, VDB will pool database connections. This can significantly reduce the load an application places on the database. Application code simply acquires and releases database connections. Instead of being freed when released, however, connections are returned to a pool of available connections managed by VDB. When the application needs a connection, it is taken from the pool. The connection pool automatically grows and shrinks with demand. (See Connection Pooling)
- **Automatic Adaptive Query Preparing and Caching.** If the same query is executed repeatedly, preparing the query ahead of time can reduce the overall execution time significantly. Because a prepared query is associated with a particular database connection, you normally don't use prepared queries as much when you use connection pooling (you don't leave queries prepared when you release a connection to the pool). VDB solves this problem by preparing and caching queries with the connection for a configurable timeout. VDB will adaptively determine when to prepare and cache queries. If the same SQL is executed multiple times within the timeout period, VDB will automatically prepare and cache the query. This means that you don't have to determine which queries should be prepared and which should not. VDB will do this automatically. This is especially useful when the pattern of code execution varies, as with library code or code that is executed heavily only at a particular time of day. (See Automatic, Adaptive Query Preparing and Caching)

1.1.2 Example

Here is an example application, which displays the name of every account in the database. First, the application is presented using the BDE directly. Then we show the same program using VDB in two different ways. The first way more closely matches the BDE example, the second uses a VDB shortcut.

BDE Example

This example shows in code all that must be configured and managed to execute a query using the BDE components. Notice that configuring the database and query objects is somewhat involved. Also, notice the use of try/finally/end to ensure that we delete these objects.

Of course, much of this can be done at design-time, but the example is still interesting. If you wanted to use connection pooling or adaptive query preparing and caching, then you'd need to do much of what is done at design-time at runtime.

```
procedure ShowAccounts_BDE;
var
  Database: TDatabase;
  Query: TQuery;
begin
  Database := TDatabase.Create(nil);
```

```
try
  Database.DatabaseName := 'TempDB';
  Database.DriverName := 'INFORMIX';
  Database.Params.Values['SERVER NAME'] := 'nike_SDTCP';
  Database.Params.Values['USER NAME'] := 'dbuser';
  Database.Params.Values['PASSWORD'] := 'dbuser';
  Database.LoginPrompt := False;
  Database.Open;

  Query := TQuery.Create(nil);
  try
    Query.SessionName := Database.SessionName;
    Query.DatabaseName := Database.DatabaseName;

    Query.SQL.Text := 'select * from acct where acct_nbr <= 5';
    Query.Open;
    while not Query.Eof do begin
      Writeln(Query['name']);
      Query.Next;
    end;
  finally
    Query.Free;
  end;
finally
  Database.Free;
end;
end;
```

VDB version most like BDE Version

The first thing you'll notice is that this is shorter than the BDE version. This is because configuring the database and query objects is simpler in VDB. Also, VDB uses interfaces, eliminating the need to explicitly free these objects.

If you take a closer look at these two examples, you will notice that they have several properties and methods in common. For example, in both versions, the database objects have the Open method, and the query objects have the following members: SQL, Open, Eof, Next, and the FieldValues array property.

```
procedure ShowAccounts_VDB_Long;
const
  Profile =
    'vdb:vdbEngine=bde;driver=INFORMIX;server=nike_SDTCP;user=dbuser;password=dbuser';
var
  Database: IDatabase;
  Query: IQuery;
begin
  Database := VDBEngines.NewDatabase(Profile);
  Database.Open;

  Query := Database.NewQuery;
  Query.SQL := 'select * from acct where acct_nbr <= 5';
  Query.Open;
  while not Query.Eof do begin
    Writeln(Query['name']);
    Query.Next;
  end;
end;
```

Even Shorter VDB Version

The VDB version can be shortened even further, using additional methods like IDatabase.RunQuery, for example, which creates the new query, sets its SQL property and opens it, all in one method call.

```
procedure ShowAccounts_VDB_Short;
const
  Profile =
    'vdb:vdbEngine=bde;driver=INFORMIX;server=nike_SDTCP;user=dbuser;password=dbuser';
var
  Database: IDatabase;
  Query: IQuery;
begin
```

```

Database := VDBEngines.NewDatabase(Profile);
Query := Database.RunQuery('select * from acct where acct_nbr <= 5');
while not Query.Eof do begin
    Writeln(Query['name']);
    Query.Next;
end;
end;

```

1.2 Connecting

In this section we will see how to configure your application to use VDB and how to connect to the database using profile strings.

1.2.1 Configuring Your Application to Use VDB

Here are the steps to create a new application and configure it to work with VDB. (Note that this sample application is a console application for simplicity; these steps apply equally well to GUI applications).

- Create a directory for your application.
- Launch Delphi and create a new application and save it in the directory you created in the first step.
- Open up the project file (Project|View Source) and add VirtualDB and VDB_BDE as the first units in the program's uses clause. That is, make the uses clause look something like:

```

uses
    VirtualDB,
    VDB_BDE,
    VDB_ADO,
    VDB_ADS,
    VDB_DBISAM,
    VDB_DBX,
    VDB_FlashFiler,
    VDB_IBX,
    ...

```

This example shows using all five built-in VDB engines. You can include any or all of these VDB engine units, depending on the needs of your application. You should only include the ones your application is actually going to use. When you want a unit in your application to use VDB, then simply put VirtualDB unit in that unit's implementation or interface uses clause (whichever is appropriate). You only need the driver unit (e.g. VDB_BDE) in the application's uses clause. Each engine/driver registers itself with VDB.

It is important that you put VirtualDB before the Forms unit (or any unit that uses Forms). There is a problem with Forms that is solved by putting VirtualDB first. The safest place to put VirtualDB is first in the uses clause.

1.2.2 Profile Strings

In order to connect to a database, you need to create an IDatabase instance. There are several ways to create an IDatabase object within VDB, but the simplest way is to use profile strings. A profile string encodes the name of the data engine and any additional parameters used by the database library. For example, a profile string for connecting to a SQL-Server database via the BDE would look something like this:

```
vdb:vdbEngine=bde;driver=MSSQL;server=Dev-Server;database=DevITS;user=sa
```

In general, profile strings have the following form:

```
vdb:vdbEngine=engine;param1=value1;param2=value2;param3=value3...
```

Engines

The engine (as specified by the vdbEngine parameter) must match one of the registered data engines. You simply need to include the appropriate engine unit (or units) in your application to make an engine available to VDB. You can only use an engine if the unit is compiled into the application (or loaded in a package). There are currently eight built-in engines:

Name	Unit	Description
BDE	VDB_BDE	<i>Borland Database Engine.</i> The traditional library for Delphi applications.

ADO	VDB_ADO	<i>Microsoft Active Data Objects.</i> Requires Microsoft ADO.
ADS	VDB_ADS	<i>Advantage Database Server.</i> Requires Advantage Database Server from Extended Systems (www.AdvantageDatabase.com).
DBISAM	VDB_DBISAM	<i>DBISAM Database Server.</i> Requires DBISAM from Elevate Software (www.ElevateSoft.com).
FlashFiler	VDB_FlashFiler	<i>Flash Filer 2.</i> Requires Flash Filer 2 from TurboPower Software (www.TurboPower.com).
IBX	VDB_IBX	<i>InterBase Express.</i> Provides access to InterBase databases; compiles right into your application.
DBX	VDB_DBX	<i>DB/Express.</i> A lightweight, cross-platform library that is available in Delphi 6 and Kylix (Delphi for Linux).
SQLDirect	VDB_SQLDirect	<i>SQL-Direct.</i> A third-party library that resembles the BDE, and supports many RDBMS systems.

Parameters

The parameters that follow the engine in the profile are used to configure the database object within the indicated engine. VDB defines several common parameters, making profile strings more uniform across data engines. The following tables summarizes them.

VDB engines for Delphi's built-in database access libraries:

VDB	BDE	ADO	DBX	IBX
Server	Server Name	Data Source	HostName	DatabaseName*
Database	Database Name	Initial Catalog	Database	N/A
User	User Name	User ID	User_Name	User_Name
Password	Password	Password	Password	Password
Driver	DriverName*	Provider	DriverName*	N/A
Alias	AliasName*	N/A	N/A	N/A
Dialect	N/A	N/A	N/A	SQLDialect*

VDB engines for 3rd-party database access libraries:

VDB	ADS	DBISAM	FlashFiler	SQLDirect
Server	N/A	RemoteHost*	Transport.ServerName*	RemoteDatabase*
Database	N/A	RemoteDatabase*	Database.AliasName*	Database Name
User	Username*	RemoteUser*	Client.UserName*	User Name
Password	Password*	RemotePassword*	N/A	Password
Driver	N/A	N/A	N/A	ServerType*
Alias	AliasName*	N/A	N/A	N/A

Dialect	N/A	N/A	N/A	SQL Dialect
---------	-----	-----	-----	-------------

The parameters marked with an asterisk (*) are actually properties of the database component (i.e., TDatabase for BDE or TIBDatabase for IBX).

In addition, VDB has parameters that are used to configure features that VDB itself supplies. These parameters all begin with "vdb".

- **vdbEngine** specifies the VDB engine. Technically, this parameter does not need to be the first one, but we recommend putting it first for increased readability.
- **vdbBaseProfile** specifies a base profile, for use with the Linked Profiles feature. This allows profiles to build on one another.
- **vdbPoolTimeout** specifies the amount of time, in seconds, that a connection will remain in the connection pool before being dropped. A value of zero (the default) means that connections will not be pooled.
- **vdbPoolName** gives a name to a connection pool. When VDB creates a connection pool, only connections with identical connection parameters can be in the same pool. Specifying a vdbPoolName can make a profile string unique that would otherwise match another profile string. You might want to use this feature if part of your application used different queries than another part and you want to make optimal use of automatic query preparing and caching. Rather than have all connections prepare and cache queries that are used by the whole application, each part would prepare and cache queries only for its smaller pool of connections. Another example would be an application server executable that hosts multiple applications. A separate pool for each distinct application may be desirable.
- **vdbQueryTimeout** specifies the amount of time, in seconds, that a query will automatically remain in the query cache that each IDatabase instance maintains. Frequently used queries are automatically prepared and cached for each connection. This can significantly improve the performance of your application. Because of its dynamic nature, automatic query caching and preparing can often yield better performance with less load on the database server than if the programmer were to explicitly decide whether to prepare and cache a query. This is because with many applications different parts of the application are active at different times of day. While a query is being frequently executed, it will be cached and prepared. When it is only infrequently executed, it won't be prepared, thus lessening the resource load on the server. By default, this value is 60 (seconds). Setting it to zero disables this feature.
- **vdbDialect** specifies the SQL dialect that VDB should use when resolving database-independent SQL. VDB will often be able to tell what dialect of SQL it should use. For example, if you use the MSSQL BDE driver, then VDB will know that you are connecting to SQL Server and that it should use the SQL Server dialect. In cases where it can't tell, you explicitly specify the dialect.
- **vdbUnidirectional** specifies the default value of the Unidirectional property of IQuery objects created by an IDatabase object. Some database libraries do not support the Unidirectional property. That is, some always have unidirectional queries while others always have bidirectional queries. In those cases, this parameter is ignored. This parameter is mainly useful when porting from a library that supports both unidirectional and bidirectional queries (such as BDE and IBX) to a unidirectional-only library (such as DBX). It may also provide better performance and scalability due to reduced in-memory row caching. The default value is False. Valid values are False and True.

For advanced configuration and profile management options see the Linked Profile Strings section of Advanced Configuration and Performance.

1.2.3 Creating a Connection

To create an IDatabase object from a Profile String call the NewDatabase method of the VDBEngines object (a singleton).

For example:

```
uses
  VirtualDB;

const
  Profile = 'vdb:vdbEngine=bde;driver=MSSQL;server=MyServer;'
    + 'database=mydb;user=dbuser;password=mysecret';
var
  Database: IDatabase;
begin
  Database := VDBEngines.NewDatabase(Profile);
  ...
```

See Profile Strings for more information.

1.3 VDB Shortcuts

In this section we will examine the shortcuts VDB provides to make your code smaller, easier to read, and more expressive. We start with simple SQL statements. We then explore parameterized queries. We then show how to use the special formatting symbols VDB provides. We complete this section with a discussion of transactions and the special facilities VDB provides for managing them.

1.3.1 NewQuery

With VDB you typically use an IDatabase object to create IQuery objects.

The IDatabase.NewQuery method that takes no parameters is the most straightforward way to create a query object. The resulting IQuery is attached to the database that created it, but none of its properties have been set. You can take the IQuery and set its SQL property, Open it, call its ExecSQL or SetParams methods, or anything else you'd like to do with it. The ShowAccounts_VDB_Long example in the introduction illustrates the parameterless NewQuery method.

All of the rest of the methods for creating and executing a query (the other flavors of NewQuery, RunQuery, QueryValue and ExecSQL) are shortcuts for creating a query with the parameterless NewQuery method and setting properties and calling methods on the resulting IQuery object, and possibly extracting some data from the IQuery object.

See Parameterized SQL and Formatting SQL sections for more versions of NewQuery.

Notes

Normally you won't call the parameterless version of this method. Not only are the other methods shorter, but they work with the automatic query caching and preparing. Every version of NewQuery, RunQuery, ExecSQL, QueryValue, etc., except for the parameterless version of NewQuery, first translate any vendor-neutral SQL (function escapes) into vendor-specific SQL and use the translated SQL to look for a cached (and possibly prepared) version of the query based on the SQL passed in. See the Automatic Adaptive Query Preparing and Caching, Function Escapes and VDB:Dialects sections for more information.

1.3.2 RunQuery

The IDatabase.RunQuery method is just like NewQuery, except that it also calls the IQuery.Open method for you. So, the following code:

```
Query := Database.NewQuery('select * from account');  
Query.Open;
```

is the same as

```
Query := Database.RunQuery('select * from account');
```

See Parameterized SQL and Formatting SQL sections for more versions of RunQuery.

1.3.3 ExecSQL

The IDatabase.ExecSQL method is the same as the NewQuery method, except that it also calls the IQuery.ExecSQL method for you and returns the number of rows affected (just like IQuery.ExecSQL) instead of returning the query itself. So the following code:

```
with Database.NewQuery('delete account where acct_nbr = 5000') do  
  Count := ExecSQL;
```

is the same as

```
Count := Database.ExecSQL('delete account where acct_nbr = 5000');
```

IDatabase.ExecSQL is used for insert, update and delete statements, and any other statement that returns the number of rows affected instead of a result set.

See Parameterized SQL and Formatting SQL sections for more versions of ExecSQL.

1.3.4 QueryValue and QueryValueDef

QueryValue

The IDatabase.QueryValue method is the same as the RunQuery method, except that it verifies that exactly one row was returned. If one row wasn't returned, then it raises an exception. It also extracts the contents of that row as a variant. If one column is returned then the value returned is the single value in the one row/column in the result as a Variant. If more than one

column is returned, then the value is a Variant Array.

So, the following code:

```
with Database.RunQuery('select x from b where id = 1') do begin
  if IsEmpty then
    raise EVDBNotASingletonError('Query did not return a singleton');
  Count := Query.Fields[1].AsVariant;
  Next;
  if not Eof then
    raise EVDBNotASingletonError('Query did not return a singleton');
end;
```

is the same as

```
Count := Database.QueryValue('select x from b where id = 1');
```

There is also a version that allows you to specify the column or columns you want returned.

QueryValueDef

The IDatabase.QueryValueDef method is the same as the QueryValue method, except that you can pass in a default value to return when the query returns no rows, instead of raising an exception.

So, the following code:

```
with Database.RunQuery('select x from b where id = 1') do begin
  if IsEmpty then
    Count := -1;
  Count := Query.Fields[1].AsVariant;
  Next;
  if not Eof then
    raise EVDBNotASingletonError('Query did not return a singleton');
end;
```

is the same as

```
Count := Database.QueryValueDef(-1, 'select x from b where id = 1');
```

There is also a version that allows you to specify the column or columns you want returned.

See Parameterized SQL and Formatting SQL sections for more versions of QueryValue and QueryValueDef.

1.3.5 Parameterized SQL Shortcuts

Overview

Parameterized SQL is SQL that contains named place-holders for values that are supplied when the SQL is executed. Parameters start with a colon and are named. For example, the following SQL contains one parameter named account_nbr.

```
delete from account where account_nbr = :account_nbr
```

When this SQL is executed, a value must be supplied for the account_nbr parameter.

There are many reasons why you'd want to use parameterized SQL. If you're inserting or updating a blob field you can't pass its value in the SQL string. You have to use a parameter. Additionally, parameterized SQL can be more efficient if it is prepared and executed multiple times, with different values for parameters of the same SQL string.

Note that you can explicitly prepare queries by calling the IQuery.Prepare method or by setting the IQuery.Prepared boolean property. However, VDB will automatically prepare and cache queries that are executed frequently, so you can use prepared queries without ever explicitly calling Prepare. See the Automatic Adaptive Query Preparing and Caching section for details.

Note that automatic query preparing and caching is another good reason to use parameterized SQL. Because only the parameter names are included in the SQL that gets prepared, the same prepared query can be used over and over again, if the SQL is the same, but only the parameter values differ. If you were to format unique SQL strings with the values in the SQL, then the same prepared query cannot be used repeatedly. The query must be prepared for each execution instead of being reused.

The following example (which will be explained below) illustrates using parameters with QueryValue:

```
WriteLn(Database.QueryValue(
  'select name from acct where acct_nbr = :acct_nbr',
  [ParamValue('acct_nbr', 1023)]));
```

In addition to a name and a value, parameters have an associated "data type" (such as string, datetime, or in the example above, integer) and a "parameter type" (such as input, output and input/output). SQL select, insert, update and delete statements only have input parameters, so the "parameter type" is always "input" for these queries. For this discussion we will only be concerned with input parameters.

TParamRec, Param and ParamValue

When an IQuery is configured to run parameterized SQL, the name, value, datatype and parameter type must be supplied. This sounds complicated, but VDB provides many facilities for easily creating and executing parameterized queries, including select, insert, update and delete queries.

VDB provides a very simple abstraction for a SQL parameter: TParamRec. TParamRec is simply a record that holds the four pieces of information required for a parameter: The Name, DataType, ParameterType (input, output or input/output) and Value.

A record is a simple abstraction (and memory managed!) but still a bit awkward to work with directly. So, VDB also provides some very simple helper functions for creating instances of TParamRec that simplify working with TParamRecs.

The most useful function for working with parameters is ParamValue. Most of the time you can simply use ParamValue and pass just the Name and Value of the parameter. In this case the DataType is inferred from the Value. If the Value (which is a Variant) holds an Integer, then the DataType is set to ftInteger. If it holds a string, then the DataType is set to ftString, etc. The ParamType is set to ptInput when using ParamValue. That's all four bits of information required for a parameter.

When you need to either change the default datatype or there is not default datatype (e.g. for NULL) then you also need to specify the DataType. For example, if you are passing Null as the Value, then you'll also need to specify the datatype because Null isn't a string, integer, datetime, etc. For another example, you might need to specify ftMemo instead of using the default ftString when passing a string as a parameter that is being inserted into the BLOB field.

VDB provides other helpful functions that are variations on ParamValue, including Param, ParamOpt, and ParamValueOpt

NewQuery, RunQuery, ExecSQL and QueryValue with TParamRec

The IDatabase interface provides overloaded versions of the IDatabase.NewQuery, RunQuery, RunQuery, QueryValue and QueryValueDef methods that take a SQL string (with embedded parameters) and an array of TParamRec objects. The example in the Overview illustrates the QueryValue method that takes an array of TParamRec. All of these methods use the SetParams method after setting the SQL and before performing any of their actions (such as calling Open in the case of RunQuery).

SetParams simply copies the values from the array of TParamRec into the Params property of the IQuery. The Params property is discussed in the Delphi online help. Look under TQuery.Params for more information. You won't have to use this function if you're using one of the flavors of NewQuery, RunQuery, ExecSQL or QueryValue that use TParamRec, but it's there if you need it.

Examples

Below are some examples using these helper functions. You'll note that NewQuery uses the Param function, while the rest use the ParamValue. This is because NewQuery isn't opening or executing the SQL given. It just configures the query. That is, the query is created outside the loop and just the parameters are set inside the loop. For the other examples, the parameters are configured and the SQL is executed in one method call, so the values must be supplied in that call.

It appears that a new query is being created in each iteration of the loop; actually, because VDB automatically caches and prepares queries that are executed repeatedly, this is not the case. In each example, only one query object is created. See Automatic Adaptive Query Preparing and Caching for details.

```
procedure NewQueryParamExample(const Database: IDatabase);
var
  i: Integer;
  Query: IQuery;
  p: TParam;
begin
  Query := Database.NewQuery(
    'select name from acct where acct_nbr = :acct_nbr',
    [Param('acct_nbr', ftInteger)]);
  p := Query.Params.ParamByName('acct_nbr');
  Query.Prepare;
  for i := 1 to 5 do begin
    p.Value := i;
    Query.Open;
```

```

        Writeln(Query['name']);
        Query.Close;
    end;
end;

procedure RunQueryParamExample(const Database: IDatabase);
var
    i: Integer;
    Query: IQuery;
begin
    for i := 1 to 5 do begin
        Query := Database.RunQuery(
            'select name from acct where acct_nbr = :acct_nbr',
            [ParamValue('acct_nbr', i)]);
        Writeln(Query['name']);
    end;
end;

procedure ExecSQLParamExample(const Database: IDatabase);
var
    i: Integer;
begin
    for i := 1 to 5 do begin
        Writeln('Rows Updated = ',
            Database.ExecSQL(
                'update acct set name = name where acct_nbr = :acct_nbr',
                [ParamValue('acct_nbr', i)]));
    end;
end;

procedure QueryValueParamExample(const Database: IDatabase);
var
    i: Integer;
begin
    for i := 1 to 5 do begin
        Writeln(Database.QueryValue(
            'select name from acct where acct_nbr = :acct_nbr',
            [ParamValue('acct_nbr', 1023)]));
    end;
end;
end;

```

1.3.6 SQL Formatting Shortcuts

The IDatabase interface provides methods NewQueryFmt, RunQueryFmt, ExecSQLFmt, QueryValueFmt, and QueryValueFmtDef that take a SQL string and an array of variants. These methods work like the built-in Format function in that they substitute the values in the array into the SQL string based on format specifiers like %s, %d, etc.

The format specifiers, however, are different from those used by Format; they are specialized for SQL. For example, the %s specifier is replaced with a quoted version of the string supplied in the parameter.

In addition, IDatabase provides a FormatSQL method that just does this substitution.

Format Specifiers

Letter	Meaning	Data Type	Result
B	Boolean	Boolean	'Y' or 'N'
D	Decimal	Integer, SmallInt	just the number
F	Float	Single, Double, Currency	just the number
L	Literal	String, OleStr	just the string
S	String	String, OleStr	the string, quoted for SQL

V	Variant	Any type in this table	based on type works like B, D, F, S, T
T	DateTime	DateTime	Date/Time, quoted for SQL, based on the SQL dialect
A	Date	DateTime	Date, quoted for SQL, based on the SQL dialect
M	Time	DateTime	Time, quoted for SQL, based on the SQL dialect

Basically, calling the Fmt version of a method is the same as calling FormatSQL first and then calling the non-Fmt version that takes just a SQL string. For example, the following code:

```
Database.QueryValueFmt(
  'select acct_nbr from acct where name = %s',
  ['ACCESS INVESTMENTS']);
```

is the same as

```
Database.QueryValue(
  Database.FormatSQL(
    'select acct_nbr from acct where name = %s',
    ['ACCESS INVESTMENTS']));
```

and results in the following SQL being executed. *Note the quoting!*

```
select acct_nbr from acct
where name = 'ACCESS INVESTMENTS'
```

You'll notice in the table above that the datetime (T) date (A) and time (M) format specifiers indicate that they are specific to the SQL dialect. This is because the SQL Server datetime format is different than the Interbase datetime format, which is differs from the Informix datetime format, etc. The FormatSQL method can distinguish between dialects and select a datetime format accordingly.

1.3.7 Transaction Shortcuts

The IDatabase object exposes many of the properties and methods of the underlying database component (TDatabase, TADOConnection, TIBDatabase, TSQLConnection, etc.). This includes the transaction-related methods StartTransaction, Commit, and Rollback.

Unfortunately, you can't nest calls to these functions. If you call StartTransaction while you're already in a transaction, an exception is raised. Delphi provides somewhat of a way out with the InTransaction method that returns a Boolean. This leads to awkward code. For example, suppose you have a utility function that executes two related updates that should be in a transaction so that they both either commit or rollback. This function may be called with a database that may or may not be in a transaction. You'd have to write the following awkward code:

```
var
  WasInTransaction: Boolean;
begin
  // Isn't this awkward?
  WasInTransaction := Database.InTransaction;
  if not WasInTransaction then
    Database.StartTransaction;
  try
    // Do work here
    if not WasInTransaction then
      Database.Commit;
  except
    if not WasInTransaction then
      Database.Rollback;
    raise;
  end;
end;
```

That's a lot of code just to manage transactions. VDB provides a solution with the IDatabase methods PushTrans, PopCommit and PopRollback. These methods nest and should be balanced. For example, the above code could be rewritten as follows:

```
begin
  Database.PushTrans;
```

```

try
  // Do work here
  Database.PopCommit;
except
  Database.PopRollback;
  raise;
end;
end;

```

The IDatabase keeps track of a count that gets incremented when PushTrans is called and decremented when either PopCommit or PopRollback is called. When the count goes from 0 to 1 StartTransaction is called. When it drops from 1 to 0 either Rollback or Commit is called, depending on whether PopRollback was called or PopCommit was called.

Because you need to balance the PushTrans call with a call to either PopCommit or PopRollback, it is typical to use the pattern above. Call PushTrans and in a try-except call PopCommit at the end of the try-block and inside the exception handler call PopRollback and re-raise the exception. This way, you'll balance the PushTrans call with a call to one of the Pop methods. You'll call PopCommit if the code in between was successful and you'll call PopRollback if an exception was raised.

Notes

Make sure you don't call Exit inside the try-block or you won't call PopCommit or PopRollback!

1.4 Vendor Neutral SQL

In this section we explore the features of VDB that make it easier to write SQL that is portable from one database to another. That is, not only can you switch from one engine to another (such as from the BDE to DBXpress) but you can also port from one database to another (such as from Informix to SQL Server).

1.4.1 Function Escapes

VDB offers a flexible way to write vendor-neutral SQL statements that contain functions. For example, suppose you want to run this query against Informix:

```

select * from acct where length(name) > 2

```

Of course, the above query returns all accounts with names longer than two characters. The sad fact is that the "length" function is not standard -- SQL Server calls it "len", while InterBase calls it "strlen". Similar problems plague the "substr", "trunc", and "ceiling" functions.

VDB solves this problem by allowing you to use a standard name for a given function. VDB will automatically translate this standard name into the vendor-specific name as required by the database you're using. For example:

```

select * from acct where [length(name)] > 2

```

Notice the square brackets -- VDB looks for text inside square brackets, and translates any function calls from the VDB standard syntax into the database-specific syntax.

In addition to simple functions like "abs", "sqrt", and "length", VDB presents other database features as functions usable within square brackets. In fact, all of the features of the dialect macros (vdbNow, IDatabase.IdentityQuery, IDatabase.AddDateTime, etc) are available as functions.

Let's take another example:

```

select * from order where ship_date > [now()]

```

This gets all orders with a shipping date after the current datef (i.e., using the RDBMS's clock). Notice that the parenthesis are required even with zero-argument functions.

Math Functions

All math functions take one numeric argument and return a numeric result.

abs	absolute value
acos	arc cosine
asin	arc sine
atan	arc tangent

ceil	ceiling
cos	cosine
exp	exponentiation
floor	floor
log10	log base 10
mod	modulus
sin	sine
sqrt	square root
tan	tangent

String Functions

concat	Concatenates two or more string arguments.
length	Takes a single string argument and returns the number of characters in the string.
lower	Takes a single string argument and returns a a string with the uppercase letters changed to lowercase.
ltrim	Left trim. Takes a single string argument and returns a string with left-whitespace removed.
rtrim	Right trim. Takes a single string argument and returns a string with right-whitespace removed.
substr	Substring. Takes a string and two integers and returns the portion of the string starting at the index of the first integer parameter with a length equal to the second integer parameter (or to the end of the string, whichever is reached first).
trim	Takes a single string argument and returns a string with whitespace stripped from both ends of the string.
upper	Takes a single string argument and returns a a string with the lowercase letters changed to uppercase.

Date/Time Functions

Note that the date and time functions are overloaded.

now, current_datetime, current_timestamp	Each of these functions takes no arguments and returns the current date and time using the RDBMS clock. These functions are equivalent.
today current_date	Each of these functions takes no arguments and returns the current date. These functions are equivalent.
timestamp datetime	Takes a string-literal that is a date-time and returns a datetime value. Any string-literal that Delphi's StrToDateTime accepts is allowed. For example, [datetime(12/31/1999 23:59:59)] becomes the datetime value a second before the year 2000. These functions are equivalent.
date	Takes a string-literal that is a date and returns a date value. Any string that Delphi's StrToDate accepts is allowed. For example, [date(12/31/1999)] becomes a day before the year 2000. These functions are equivalent.

format_datetime, format_timestamp	Takes a string-literal that is a date-time and formats it for the RDBMS. Any literal string that StrToDateTime accepts is allowed. Typically, the "datetime" or "timestamp" functions are used, but if you need a string value instead of a datetime value, then use these functions.
format_date	Takes a string-literal that is a date and formats it for the RDBMS. Any literal string that StrToDate accepts is allowed. Typically the "date" function is used, but if you need a string value instead of a date value, then use format_date.
adddatetime	Takes 3 arguments: a datetime expression, an integer and a date-time part. For example: [AddDateTime(now(), 1, DAY)] becomes an expression that yields one day from now, using the RDBMS clock. The valid "date-time parts" are: YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND. The plurals of these are also allowed (e.g. YEARS).

Misc Functions

identity	Takes no arguments and returns a select statement that returns the most recent auto-increment value inserted by the current connection into a table with an auto-increment column (also known as serial and identity columns). This can be used to retrieve the value of a serial column in a table immediately after an insert into that table. Note that some databases (DB2) require that the insert take place inside an explicit transaction that is not yet committed or rolled back.
option	Takes 1 argument, which is the name of an option defined in the database object. Options are application-specific settings that can be accessed directly in queries. Options can be defined in a database in two ways: programmatically (using the IDatabase.Options property), or through the profile string (using the "vdbOption:name=value" syntax). The option function expands to the value of the named option. For example, if the profile string contains "vdbOption:chg_time=change_datetime", then the database could run a query as follows: select * from order_detail order by [option(chg_time)] In the above example, the <i>chg_time</i> option is used to hide the name of the column that contains the timestamp of the row. This would be useful in a library that needed to work with several different database schemas, where the name of this column was not consistent. For example, the profile string for another database might define the <i>chg_time</i> option differently. This can lead to more reusable code.
ping	Takes no argument and returns a select statement that is a very simple statement that returns a small result set. This is used by IDatabase.Ping to verify that the RDBMS is up and the connection to it is functioning. While you may use this function in your SQL, you would typically use the IDatabase.Ping method instead. It executes the ping query and if that fails, closes and re-opens the connection and tries again. If the second attempt fails, then it raises a "Database Down" exception. NOTE: Some primitive SQL dialects offer no reasonable way to implement ping without knowing the name of at least one table in the database. In these cases, the ping query will require the option (see above) "singleton" to name a table (ideally, one with a single row in it). For example, Advantage, FlashFiler, and DBISAM all require the "singleton" option in order for ping to work.
scope	This function allows you to override the default schema (<i>i.e.</i> , database) and/or owner for a table or other object. There are two overloaded versions of this function: [scope(Database, Object)] and [scope(Database, Owner, Object)] In the rare case where you want to override the owner, but not the database, you may leave the first argument blank, or use an asterisk. Example 1 - the following query selects from the customer table in the dev database: select * from [scope(dev, customer)] In Informix, the above becomes... select * from dev::customer In SQL Server and Sybase, it would be... select * from dev..customer Example 2 - the following query selects from the customer table owned by the dba user, in the dev database: select * from [scope(dev, dba, customer)] In Informix, the above becomes... select * from dev::dba.customer In SQL Server and Sybase, it would be... select * from dev.dba.customer Example 3 - the following query selects from the customer table owned by the dba user, in the default database: select * from [scope(*, dba, customer)] In Informix, SQL Server and Sybase, the above becomes select * from dba.customer

Nesting

Not only can you call these functions individually, but you can use multiple functions in an expression. Here's an example:

```
select * from order
where ship_date > [AddDateTime(now(), 1, DAY)]
```

The above example gets all orders that ship after tomorrow. Notice that we are using two VDB functions here: AddDateTime()

and now()). For Informix, VDB converts this query into:

```
select * from order
where ship_date > (current +1 units DAY)
```

But on SQL Server it would be...

```
select * from order
where ship_date > dateadd(DAY, +1, getdate())
```

Using SQL Within Functions

Within the square brackets that delimit a VDB function escape, it is sometimes necessary to prevent VDB from attempting to parse certain text. For example, consider the following query:

```
select * from cust
where cust_id = abs(
select max(cust_id) from cust)
```

In order to use the [abs()] VDB function, we must enclose the entire call to abs() in square brackets, as follows:

```
select * from cust
where cust_id = [abs(
select max(cust_id) from cust)]
```

The problem above is that VDB will attempt to resolve max(cust_id) as a VDB function, which will fail. The solution is to escape the argument to [abs()] with the backwards accent (`), as follows:

```
select * from cust
where cust_id = [abs(
`select max(cust_id) from cust`)]
```

The backwards accent characters delimit text that should be passed through verbatim to the final SQL statement.

Using SQL Functions in Your Queries

Before opening a query, VDB looks for square brackets (ignoring them if they appear in quoted strings, obviously), performing any necessary translations. Before you open the query, the SQL may contain square brackets, but after you open it, it will not (instead, it will contain the vendor-specific syntax). If, for some reason, you wish to expand the functions before opening the query, you can use the TranslateSQL method of the query.

This feature works fine with the automatic query perparing and caching feature. The database object maintains a cache of recently-executed queries (identified by the vendor-specific syntax).

If, for some reason, you wish to translate a SQL statement without even creating a query object, you may use the IDatabase.TranslateQuery method (or IDialect.TranslateQuery, if you don't want to create a database).

Finally, to determine whether a database supports a specific function (before attempting to run the query), you can use the SupportsFunction method of IDatabase (or IDialect).

1.4.2 Dialects

VDB provides flexible mechanisms to write vendor-neutral code, such as SQL statements that contain functions that are portable. In order to do so, VDB needs to know the dialect of the server it is connected to. In many cases, VDB can determine the vendor, and, therefore, the dialect of SQL being used. For example, if you are using the BDE with the MSSQL driver, then VDB will automatically set the dialect to SQLServer. In cases where VDB cannot automatically determine the dialect, then you must specify the dialect (typically in the Profile String) in order to use features like SQL function escapes.

Dialects are modelled by the IDialect interface. VDB keeps a global collection of dialects by name in the VDBEngines singleton. Currently, Access, DB2, Informix, Interbase (both 5 and 6), MySQL, Oracle (both 8 and 9), Paradox, SQLServer, and Sybase are supported. You may implement IDialect yourself and register your implementation with the VDBEngines singleton.

1.5 Advanced Configuration

In this section we examine the advanced performance and configuration features of VDB.

1.5.1 Connection Pooling

VDB will pool database connections if the profile string indicates that pooling should be used (vdbPoolTimeout). This is particularly useful in a multi-threaded program such as a server in an n-tier application or in a web application where connection pooling can significantly reduce the number of database connections. Reducing the number of connections will reduce the load

on the database. For example, if a server application had 100 clients connected to it, but there were at most 10 client requests executing in the server at the same time, then there would be at most 10 simultaneous connections to the database. During periods of inactivity, the number of connections will drop (possibly to zero). During periods of high activity, the number of connections would rise accordingly.

Connection pooling in VDB is mostly "under the hood". You don't have to do much to enable and use connection pooling. It is important, however, to understand what the implications are because it changes the way you code.

The main thing to understand is that when you attempt to create a database connection, instead, you might be getting an already connected `IDatabase` object off of a pool maintained by VDB. Only if there are no connections available in the pool will a new connection actually be created. In addition, when you release a database connection you aren't closing the connection. VDB will automatically put the connection on a pool to be re-used. If the connection is not reused before the timeout elapses, then the connection will be closed and the `IDatabase` will be discarded.

This has some consequences for how you write code. Most importantly, you shouldn't hold onto a connection. You should create a connection when you need one and release it as soon as you're done with it. If you hold onto a connection when you aren't using it, you defeat the purpose of connection pooling (sharing the connection with other operations / threads). In addition, you should release the database in the same state you received it in. VDB will clean up some changes you make to the database. For example, if you start a transaction, but don't roll it back or commit it, VDB will roll it back before putting the connection back on the pool. This means that whenever you get a connection from a call to `NewDatabase`, you will be getting a connection that is not in a transaction already.

Connection Pooling Parameters

- **vdbPoolTimeout.** Number of seconds. The length of time connections will be kept in the pool. This parameter is required for pooling. If there is a pool timeout > 0 then connections will be pooled. If this parameter isn't supplied (or is zero) then connections won't be pooled. The `IDatabase.PoolTimeout` property can be used to control pooling on an individual `IDatabase` object directly.
- **vdbPoolName.** This optional parameter has no purpose except to help make a VDB profile string unique. Because there is a separate pool of connections for every unique set of configuration parameters, this parameter simply helps make one profile string different from another. Using this parameter, two connection strings can otherwise be identical and their associated connections will be pooled separately.

An application uses connection pooling by setting the `vdbPoolTimeout` parameter in the Profile String (or by setting the `PoolTimeout` property of the `IDatabase` object). The timeout is in seconds and it refers to the amount of time that a connection will remain in the pool before being closed.

Example

The following example uses connection pooling, with a timeout of five minutes. If it were called repeatedly with a delay of less than 5 minutes in between, then only the first call to `NewDatabase` would cause a new connection to be created. This connection would be placed on a pool maintained by VDB. The subsequent calls would get the connection off of the pool.

```
procedure ShowAccounts_Pooled;
const
  Profile =
    'vdb:vdbEngine=bde;driver=INFORMIX;server=nike_SDTCP;'
    + 'user=dbuser;password=dbuser;'
    + 'vdbPoolTimeout=300'; // 300 seconds = 5 minutes
var
  Database: IDatabase;
  Query: IQuery;
begin
  Database := VDBEngines.NewDatabase(Profile);
  Query := Database.RunQuery(
    'select * from acct where acct_nbr <= 5');
  while not Query.Eof do begin
    Writeln(Query['name']);
    Query.Next;
  end;
end;
```

1.5.2 Automatic, Adaptive Query Preparing and Caching

When you execute the same SQL multiple times (often parameterized, in a loop) it is more efficient to prepare the query before

executing it and to unprepare it afterwards. For example, if you have a loop that inserts 10 records using the same parameterized SQL, it would be a lot quicker to call the `IQuery.Prepare` method before the loop and to call `IQuery.UnPrepare` afterwards.

Unfortunately, you don't always want to call the `IQuery.Prepare` method. When a query is prepared, the query sends the SQL to the database so that the database can parse the SQL and decide how it is going to execute it. It decides which indices it is going to use, etc. The database then holds onto this "query plan" until the query is unprepared (or the database connection is closed). This is a load on the database machine. The more prepared queries it has to keep track of, the greater the load.

So, sometimes you want to prepare a query and sometimes you don't. When you have a loop it is obviously beneficial. Often, however, it is difficult for the programmer to determine whether preparing a query will be a good idea or not. For example, if there are times of day when a certain query is executed frequently, but other times of day it is executed infrequently. It might make sense to create a query, prepare it (and hold onto it) while it is being executed frequently, but when it is no longer being executed so frequently, it might make sense to unprepare it (and even release it).

The situation gets even more complicated when you use Connection Pooling. Normally, once you prepare a query, you have to hold onto it (i.e. you can't keep a query prepared if it is destroyed!). Queries are associated with a connection. This means that you'd have to hold onto the associated `IDatabase` as well. But that defeats connection pooling! While one part of your application is holding onto a connection, another part can't use it. If it hasn't been released to the pool then another part of your application that is calling `NewDatabase` won't find it in the pool.

Fortunately, VDB provides a solution. When you release an `IQuery` object, VDB will put the query in a cache in the associated `IDatabase` object. Later, when you execute SQL through one of the `IDatabase` methods: `RunQuery`, `ExecSQL`, and `QueryValue`, VDB will first look in the cache to see if it has a cached `IQuery` with the same SQL. If it does, then it will first prepare the query and use it and then put it back in the cache. This way, the first time you execute some SQL, it won't get prepared, but the second and subsequent times you execute the exact same SQL (including parameter names) it will be prepared. It looks like you're creating and executing a brand-new `IQuery` object, but you're using a cached and prepared one instead.

Of course, you don't want these queries to stay around forever. So, VDB has a timeout for queries in the cache. By default, the query cache has a timeout of one minute. You can set the timeout in the database profile string with the `vdbQueryTimeout` paramter. For example, the following string sets the database's query timeout to two minutes:

```
vdbQueryTimeout=120
```

This sets the `IDatabase.QueryTimeout` property. When a query is created with one of the `IDatabase` methods: `NewQuery`, `RunQuery`, `ExecSQL`, or `QueryValue` this is the default value for the `IQuery.Timeout` property. The `QueryTimeout` property controls the timeout of the individual query. Normally, you'd either use the default value of one minute or set a value in the Profile String.

If you set a query's `QueryTimeout` to 0 then the query won't be cached. That is, when you release it, it will be freed.

In addition to this timeout, VDB sets an upper limit on the size of the query cache. It will keep at most 100 unprepared queries and at most 100 prepared queries in its cache. When this limit is exceeded, the least-recently-used query is discarded.

Why Parameterized Queries Are Preferred

Because parameterized queries don't contain values for their parameters in their SQL, but instead just have the parameter names, parameterized queries are more likely to be prepared and re-used. For this reason, parameterized queries are preferred over dynamically generated SQL strings.

For example:

```
for i := 1 to 1000 do
  Database.ExecSQL(
    'insert into some_numbers values (:some_number)',
    [ParamValue('some_number', i)]);
```

is preferred over:

```
for i := 1 to 1000 do
  Database.ExecSQLFmt(
    'insert into some_numbers values (%d)', [i]);
```

The first example will take advantage of automatic query preparing and caching. The second example, while still functional, will not take advantage of this feature. The second through 1000th insert in the first example will be executed through an already-prepared query. In the second example, each insert is executed through its own query object, without first being prepared.

See Parameterized SQL for more information.

1.5.3 Linked Profile Strings

Many applications define a set of profiles, selecting a specific profile at run-time. For example, you might keep the profiles in the registry or an INI file, then select one based on a command-line option. In this situation, it can be cumbersome to define many profiles that differ only slightly. Consider the following examples:

```
SalesDev="vdb:vdbEngine=BDE;driver=MSSQL;server=Dev;database=sales;
user=dbuser;password=mysecret;vdbPoolTimeout=600"
SalesTest="vdb:vdbEngine=BDE;driver=MSSQL;server=Test;database=sales;
user=dbuser;password=mysecret;vdbPoolTimeout=600"
SalesProd="vdb:vdbEngine=BDE;driver=MSSQL;server=Prod;database=sales;
user=dbuser;password=mysecret;vdbPoolTimeout=600"
AcctDev="vdb:vdbEngine=BDE;driver=MSSQL;server=Dev;database=acct;
user=dbuser;password=mysecret;vdbPoolTimeout=600"
AcctTest="vdb:vdbEngine=BDE;driver=MSSQL;server=Test;database=acct;
user=dbuser;password=mysecret;vdbPoolTimeout=600"
AcctProd="vdb:vdbEngine=BDE;driver=MSSQL;server=Prod;database=acct;
user=dbuser;password=mysecret;vdbPoolTimeout=600"
```

As you can see, we have defined one profile for each combination of the sales and accounting databases for development, test and production systems. VDB supports linked profiles, which allow you to define one profile based on another. Here's how we could re-write the above profiles:

```
@Common-Base="vdb:vdbEngine=BDE;driver=MSSQL;
user=dbuser;password=mysecret;vdbPoolTimeout=600"
@Sales-Base="vdb:vdbBaseProfile=@Common-Base;database=sales"
@Acct-Base="vdb:vdbBaseProfile=@Common-Base;database=acct"
SalesDev="vdb:vdbBaseProfile=@Sales-Base;server=Dev"
SalesTest="vdb:vdbBaseProfile=@Sales-Base;server=Test"
SalesProd="vdb:vdbBaseProfile=@Sales-Base;server=Prod"
AcctDev="vdb:vdbBaseProfile=@Acct-Base;server=Dev"
AcctTest="vdb:vdbBaseProfile=@Acct-Base;server=Test"
AcctProd="vdb:vdbBaseProfile=@Acct-Base;server=Prod"
```

As you can see, the first profile defines the basic information that is common to all profiles. Then we define two intermediate profiles, @Sales-Base and @Acct-Base, which establish the database name. Note that we do not specify a VDB engine name in these profiles, but instead mention the base profile (with the vdbBaseProfile parameter). This informs VDB that these profiles are based on the @Common-Base profile, with additional parameters. In this way, it is much easier to change the common properties of the profiles. For example, we could change the username/password for all of these profiles just by changing @Common-Base. Finally, we define the SalesDev, SalesTest, SalesProd, AcctDev, AcctTest and AcctProd based on @Sales-Base and @Acct-Base, using the same linking mechanism.

VDB provides several functions that help manage linked profiles. The following routines resolve linked profiles:

- LoadProfileFromIniFile
- LoadProfilesFromIniFile
- LoadProfileFromRegistry
- LoadProfilesFromRegistry

The above routines rely on a convention (used in the above example) where profiles that are used to define other profiles begin with an @-sign. Profiles that are not used to define other profiles, but are "leaf-nodes" in the tree of profiles do not have the @-sign. In this way, the routines that load profiles from the registry know which profiles are intermediate profiles and which can be used to create an IDatabase object. That is, @Sales-Base isn't used to create a database object, but SalesProd is.

1.6 N-Tier Development

In this section we illustrate how to use VDB in a distributed application. This is the preferred way to use data-aware controls with VDB (the client application uses TClientDataset components, and the server uses VDB).

1.6.1 VDB Remote Datasets

In order to use remote datasets with VDB, you must inherit your remote data module from TVDBRemoteDataModule (which in

turn inherits from TRemoteDataModule). Add the unit VDBRemoteDataMod.pas to your MIDAS server project, and then use the File | New | Other | <application-name> command and select VDBRemoteDataModule.

Now that you have a TVDBRemoteDataModule, you may begin placing dataset providers on the data module. You must use TVDBProvider components, which inherit from TDataSetProvider. These provider components add special features that delay the creation of the dataset until runtime.

Rather than associate a dataset with each provider at design time, you will instead create a handler for the OnGetDataset event. In this event handler, you will perform the database access necessary to create the desired dataset. Here is an example:

```
procedure TVDBTestServer.FishProviderGetDataSet(
  Sender: TVDBProvider; out DataSet: IDataset);
begin
  DataSet := Database.RunQuery('select * from biolife');
end;
```

As this example illustrates, the event handler should assign a value to the *DataSet* parameter. This example assumes that there is a private method, *Database*, that returns a database connection. For simple applications, this method can simply call VDBEngines.NewDatabase. However, if connection pooling is required, special care must be taken to ensure that the database object is released at the end of each remote method call. For more information, see Connection Pooling in Distributed Applications

1.6.2 Connection Pooling in Distributed Applications

When using connection pooling in a distributed application server, VDB connections may be stored in a transactional context. The unit DSContext implements a per-thread collection of named resources. These resources may be transactional (as in the example of VDB connections).

The TVDBRemoteDataModule automatically supports this context mechanism. To take advantage of it in your data module, you should follow these steps:

- 1) Override the ContextFactory method to populate the context with resources such as database connections.
- 2) Call the EnterRequest method at the top of each remote method.
- 3) Balance each call to EnterRequest with one of the following methods: LeaveRequest, CommitDatabases, or RollbackDatabases.
- 4) Use the functions in VDBContext.pas to obtain database connections.

The ContextFactory method returns an ITransContextFact object, which is a factory for creating resources that will then be inserted into the context. The simplest way to implement ContextFactory is to use the NewContextFact function. Here is an example that adds a single database profile to the context:

```
function TVDBTestServer.ContextFactory: ITransContextFact;
begin
  Result := NewContextFact([
    ContextItem(tsOptional, 'MainDB', NewDatabaseFactory(sProfile))]);
end;
```

The function ContextItem takes three parameters:

- TransKind (TDSTransKind): This enumeration controls whether transactions for this resource are optional or required.
- Name (string): A unique name that identifies this resource in the context.
- Value (Variant): The resource itself, or an instance of IObjectFactory, for resources that should be created "on-demand".

In the above example, we are defining a single resource, named 'MainDB', that has optional transactions. This means that it is up to the calling code to determine whether a transaction is required -- more on this later. We are using the NewDatabaseFactory function (defined in VirtualDB.pas) to create a database connection "on-demand". This is very important because it means that idle remote data modules do not consume an active database connection.

Now that the context is configured, we can use the routines in VDBContext.pas to obtain a database connection. For convenience, we will define two private helper functions:

```
function TVDBTestServer.Database: IDatabase;
begin
```

```
    Result := GetDatabase(tsRequired, 'MainDB');
end;

function TVDBTestServer.ReadOnlyDB: IDatabase;
begin
    Result := GetDatabase(tsOptional, 'MainDB');
end;
```

The Database method returns a database connection that is guaranteed to be in a transaction. The ReadOnlyDB method returns a connection that may or may not be in a transaction. In a method that performs read-only access to the database, we can use this method to avoid unnecessary transactions.

The following examples illustrate how these methods may be used. First, we consider a dataset provider:

```
procedure TVDBTestServer.FishProviderGetDataSet(
    Sender: TVDBProvider; out DataSet: IDataset);
begin
    DataSet := ReadOnlyDB.RunQuery('select * from biolife');
end;
```

Because we are using ReadOnlyDB (as opposed to Database) in this example, the database will not necessarily be in a transaction. This is the desired mode when the provider is not resolving directly to the dataset. However, if the provider does resolve to the dataset, then we would need to request a live query, as in the following example:

```
procedure TVDBTestServer.FishProviderGetDataSet(
    Sender: TVDBProvider; out DataSet: IDataset);
var
    Query: IQuery;
begin
    Query := ReadOnlyDB.NewQuery('select * from biolife');
    Query.RequestLive := True;
    Query.Open;
    DataSet := Query;
end;
```

If your remote data module includes any remote methods (defined in the type library), then you must manage the context explicitly. There are two ways to do this, depending on whether the method needs a transaction or not. The following example illustrates how to perform read-only access to the database:

```
function TVDBTestServer.GetLargestFish: WideString;
begin
    EnterRequest;
    try
        Result := ReadOnlyDB.QueryValue('
            + ' select Common_Name from biolife'
            + ' where Length_In = (select max(Length_In) from biolife)');
    finally
        LeaveRequest;
    end;
end;
```

If the remote method makes any changes to the database, it should use a transaction:

```
procedure TVDBTestServer.SetFishName(
    SpeciesNo: Integer; const Name: WideString);
begin
    EnterRequest;
    try
        Database.ExecSQL(
            'update biolife set Common_Name = :name ' +
            'where biolife."Species No" = :species', [
                ParamValue('name', Name),
                ParamValue('species', SpeciesNo)]);
        CommitDatabases;
    except
        RollbackDatabases;
        raise;
    end;
end;
```

To summarize, for read-only methods: use EnterRequest and LeaveRequest to manage the context; use ReadOnlyDB to obtain

a database connection. For methods that update the database: use EnterRequest and CommitDatabase or RollbackDatabases to manage the context; use Database to obtain a connection.

If you follow these conventions, your remote methods will release and even commit/rollback the database connections they use. Furthermore, these remote methods may call one another, as well as other objects that use the context.

Index

A

Advanced Configuration 14
Automatic, Adaptive Query Preparing and Caching 15

C

Configuring Your Application to Use VDB 3
Connecting 3
Connection Pooling 14
Connection Pooling in Distributed Applications 18
Creating a Connection 5

D

Dialects 14

E

Example 1
ExecSQL 6

F

Function Escapes 11

I

Introduction 1

L

Linked Profile Strings 17

N

NewQuery 6
N-Tier Development 17

O

Overview 1

P

Parameterized SQL Shortcuts 7
Profile Strings 3

Q

QueryValue and QueryValueDef 6

R

RunQuery 6

S

SQL Formatting Shortcuts 9

T

Transaction Shortcuts 10

V

VDB Remote Datasets 17
VDB Shortcuts 6
Vendor Neutral SQL 11
Virtual Database 1

