**March 1998**

# Datasoft® ODBCExpress™ White Paper

*Programming ODBC: An Introduction*

# *Contents*



 

# *Introduction*

The main objectives of this white paper is to provide Delphi and C++Builder users with a clear, high-level understanding of the Open Database Connectivity (ODBC) architecture as well as the context in which ODBCExpress fits into this architecture. The ultimate aim is to provide existing and potential users of ODBCExpress with a solid understanding of how ODBCExpress works.

Most Delphi or C++Builder users with little or no concept of the ODBC SQL-oriented programming model find it difficult to move from a non-SQL-oriented programming model such as the Borland Database Engine (BDE) to ODBCExpress. This white paper will therefore also attempt to highlight the fundamental differences between ODBCExpress and the BDE.

# *What is ODBC?*

ODBC is an acronym for 'Open DataBase Connectivity'. It is a standard first developed by Microsoft for accessing data in both relational and non-relational database management systems, based on the X/Open and ISO/IEC Call-Level Interface (CLI) specifications for database APIs and the Structured Query Language (SQL) specification for database access languages. The ODBC standard is widely used by all major database vendors. Databases driven by database engines include Oracle, Microsoft SQL Server, Sybase, Informix and DB2, while you also get a lot of so-called 'flat-file' databases with no stand-alone database engines, such as Microsoft Access, dBase, Paradox and BTrieve. All these and other databases can be accessed via ODBC.

Why was ODBC created? You need a way to communicate with a database and transfer data to and from a database, as well as perform some queries on the data in the database. The problem is that each database vendor has his own set of methods (called an API) to perform all these functions. So, if you write an application which for example uses the Oracle API, that same application can't be used to access a SQL Server database, since the API's of Oracle and SQL Server are different. So you have to duplicate all your code if you want to access two different databases from the same application. Imagine if you want to access 5 or 10 different databases!

Now this is where ODBC comes into the picture and makes life a whole lot easier for you. ODBC is an API that allows you to talk to any database you want. Instead of writing a data access application using a specific database API, such as Oracle's, you can now write your application using the ODBC API, which can perform virtually any function that you can do with the Oracle API. The best part is that now you can use the same code to access any other database too! How is this possible? Well, ODBC is basically a very thin layer between your application and the database API. It makes use of an ODBC driver to access a specific database. An ODBC driver is basically a DLL which maps each function in the ODBC API to similar functions in the API of the specific database the driver was written for. For example, the Oracle ODBC driver maps the ODBC API into the Oracle API. Each database has its own ODBC driver or drivers, written either by the database vendor or a third-party ODBC driver vendor.

# *What is ODBCExpress?*

Even though you now only have to write to one API to access any database, the ODBC API isn't a simple API by any means. It has to cater for almost any functionality that any database out there can support, so you can imagine that the ODBC API is quite complex. ODBCExpress wraps the ODBC API into a set of easy-to-use classes for Delphi and C++Builder, so that you don't even have to know much (if anything) about the low-level ODBC API to fully make use of ODBC in your applications! However, the application programmer still has access to all low level ODBC functions, giving the programmer the flexibility of the ODBC API.

ODBCExpress provides you with access to the entire ODBC API. Source code availability gives you full control over the database middleware and allows you to trace down as far as the low-level ODBC call. This transparency means you can avoid the pains of impenetrable database engines during debugging. In the BDE for example ODBC calls are hidden by the BDE, which adds another layer adding complexity and problems to your database interface. So if you for example hit a problem when using BDE-ODBC you have no idea if the problem lies with the BDE or with your ODBC driver.

ODBCExpress compiles completely into your executable, which means no distribution of an external database engine is necessary. You will need to distribute the ODBC driver or drivers used by your application, and of course you need ODBC installed on the client's machine, however this is normally distributed with the operating system, and is also freely available for download from various web locations. Because ODBCExpress talks directly to ODBC, and ODBCExpress's architecture is optimized for performance, there is no other database middleware for Delphi which can match the speed of ODBCExpress. ODBCExpress was designed for the express purpose of writing large and responsive industrial-strength data access applications.

# *ODBC Architecture*

The ODBC Architecture consists of four main components:

- The Application calls ODBC functions to submit SQL statements and retrieve results.
- The ODBC Driver Manager loads and unloads ODBC drivers on behalf of the application. It also processes ODBC function calls from the application or passes them to an ODBC driver for processing.
- The ODBC Driver processes ODBC function calls, submits SQL requests to a specific Data Source and returns results to the application. If necessary the driver will modify the application's request to syntax supported by the associated database.
- The Data Source consists of the data the application wants to access and the associated operating system, DataBase Management System (DBMS) and network platform (if any) used to access the database. The ODBC Administrator (which is not a required part of ODBC when you run the application) allows you to set up and manage Data Source Names (DSNs or DataSources), each identifying a Data Source and its properties.

In other words you need the ODBC Driver Manager and the appropriate ODBC drivers installed on the machine running the application to access the appropriate databases via ODBC.

ODBC interaction from an application are done via three types of handles:

- The Environment handle is used to establish an ODBC environment for your application, by setting up a communication link from your application to the ODBC Driver Manager. You normally only need one of these handles in your application.
- For each Environment handle, you can have a number of Connection handles in your application. Each Connection handle gives you a different connection to a database. You can have multiple Connection handles to the same database or multiple Connection handles to multiple databases in your application. It is at this level at which transactioning is done and at which transaction isolation is managed.
- For each Connection handle, you can have a number of Statement handles. These handles allow you to manipulate the data in your database. Because you can have multiple Statement handles for each Connection handle, each Statement handle can perform a different function at the database and can be in a different state than the other Statement handles.

After you have all the handles you need, you can use them together with the ODBC functions in the ODBC API to make calls to the ODBC Driver Manager. And manipulate data in a single or multiple databases.

# *Implementation*

Now let's take a more in-depth look at the ODBC architecture by investigating how ODBCExpress implements it.

# *ODBC Constants and Functions*

The ODBC API consists of a number of constants and functions. In ODBCExpress the constants are contained by the ODBC Call-Level Interface Header (OCIH) unit. The ODBC functions are contained by the ODBC Call-Level Interface (OCI) unit. To hide the complexity of the ODBC calls from the ODBCExpress user, the ODBC Class Library (OCL) unit defines a number of classes which make use of the ODBC constants and functions.

# *Establishing an ODBC Environment*

An ODBC environment (of which you only need one per application) is automatically established for your application by ODBCExpress. The `THenv` environment component (a class defined in the OCL) encapsulates the environment handle (the communication link between your application and the ODBC Driver Manager) and all other properties and methods associated with the environment handle. ODBCExpress automatically creates the global variable `GlobalHenv` (located in the OCL) needed by any ODBC-enabled application. For BDE users, the `THenv` component can roughly be equated to the `TSession` component, which manages a database session for your application. As for the `THenv` component, a default `TSession` component is also automatically created for your BDE-enabled application and also only one is really needed by your application.

# *Error Handling*

As with any other API, ODBC includes error handling methods. An ODBC driver returns two sets of error codes:

- The database-specific error codes.
- A standard set of ODBC error codes.

The standard error codes (or ODBC states) give applications a standard way of dealing with errors. The ODBC driver maps each database error code to one of these standard error codes, and passes both to the application (if requested) when an ODBC function fails. But how do you determine if an ODBC function failed? An ODBC function returns a so-called "return code" which indicates if the function executed successfully or not. This return code together with the ODBC handle on which the error occurred is then used to determine the database error code, the ODBC error state, and the default error message from the ODBC driver.

ODBCExpress defines two classes to handle ODBC errors and map them to the standard Delphi exception environment. The `TODBCError` class allows you to check the validity of a return code from an ODBC function, and raise an exception if necessary. The exception raised is the `EODBC` exception class defined by ODBCExpress. This exception can be trapped using a try-except block and its properties referenced to retrieve the database error, ODBC state and error message values. Because only one `TODBCError` class is needed per application, this class is automatically created for you and can be accessed via the `Error` property of the `GlobalHenv` class. For example, say we want to execute the `SQLCancel` ODBC function:

```
RetCode:= SQLCancel(Hstmt.Handle);
```

where `RetCode` is the return code from the function, `Hstmt` is the class containing the statement handle accessed via the `Handle` property of the `Hstmt` class. Then we use the `GlobalHenv` class to check the return code and raise an error if necessary:

```
if not GlobalHenv.Error.Success(RetCode) then
    GlobalHenv.Error.RaiseError(Hstmt, RetCode);
```

We use the `RaiseError` method of the `TODBCError` class to raise an `EODBC` exception. The `RaiseError` method will query the ODBC driver for the error information given the return code and the handle class (`Hstmt` in this case) which caused the exception, and then create and raise an `EODBC` exception class. This exception can then

be trapped in a try-except block if necessary (otherwise it will be trapped by the global Delphi error handling routines):

```
try

  //code above

except
  on E: EODBC do
    ShowMessage(E.Message);
end;
```

Normally you would let the default error handler display the error message, however if you wish to test for a specific ODBC state or database error to perform some specific action, you will need a try-except block.

## *Connecting to a Database*

Now let's see how ODBCExpress establishes a connection with a database. The `THdbc` database connection class (another class defined in the OCL) encapsulates an ODBC connection handle which allows you to communicate with a specific database through ODBC. For BDE users this component can be compared with the TDatabase BDE component, which is also used to establish persistent connections with a database (in ODBCExpress all connections are persistent, in other words they won't be dropped unless instructed to do so by the application programmer, whereas the BDE also allows non-persistent connections which are automatically dropped when all result sets are closed). When a connection is established, the ODBC Driver Manager loads the appropriate ODBC driver into memory (if not in memory already because of a previous connection made through the specific ODBC driver) and instructs the driver to establish a connection with the appropriate database. But how does the ODBC Driver Manager knows which ODBC driver to load, which database to connect to and what login details to use? All this information has to be provided to the ODBC Driver Manager when wanting to connect to a database.

There are a number of different ways to provide these details. The most common way is to create an ODBC DataSource (or DSN) which contains all this information and then connect to the database via the DataSource. Such a DataSource can be created using the ODBC Administrator utility shipped with ODBC, or the DataSource can be created programmatically from your application before using it to establish the connection. The second way used to connect to a database is by establishing a DataSource-less connection, passing all the details required to establish the connection directly to the ODBC Driver Manager without first creating a DataSource containing these details.

Now which is the better way, making DataSource or DataSource-less connections? Both have their advantages and disadvantages. If you use a DataSource to establish your connection, it means the connection information is not hard-coded into your application and can be modified using a utility such as the ODBC Administrator without re-compiling your application. If you don't use a DataSource to establish your connection, the connection information is somewhat hard-coded, however this method don't require you to manually or programmatically set up a DataSource, and avoids anyone from fiddling with your connection information.

DataSources can be registry based or file based. Registry based DataSources can be system wide (seen by all users on a machine) or user wide (only seen by a specific user on a machine). When registry based, only users on the current machine can make use of the DataSource (whether it's system wide or user wide). When file based, the file containing the DataSource information can be placed on a network path and can be accessed by multiple users. Updating the file will then affect all the users making use of it. Another way of achieving this in a client/server environment is to do a DataSource-less connection, retrieving the connection details from the server side.

## *DataSource Connections*

Suppose we've set up a DataSource to an Oracle database. The DataSource defines the database server and database to connect to, so all we need to do is to provide the login details to connect:

```
Hdbc.DataSource:= 'dsnOracle';
Hdbc.UserName:= 'scott';
Hdbc.Password:= 'tiger';
```

To connect we then just call the `Connect` method of the class:

```
Hdbc.Connect;
```

If the connection fails (e.g. because the login details were wrong) an EODBC exception will automatically be raised.

## *DataSource-less Connections*

Let's connect to a Microsoft SQL Server database without using a DataSource. First we specify the database server and database to connect to:

```
Hdbc.Attributes.Clear;
Hdbc.Attributes.Add('SERVER=MyDBServer');
Hdbc.Attributes.Add('DATABASE=MyDB');
```

The `SERVER` and `DATABASE` keys are MS SQL Server specific. Different databases require different connection information (or attributes) before a connection can be established, so you will need to reference your database or ODBC driver documentation to see which keys can be used together with the `THdbc.Attributes` property. Now we need to specify the login details:

```
Hdbc.UserName:= 'sa';
Hdbc.Password:= 'mypw';
```

The `UserName` and `Password` properties are common to all databases, therefore they are provided as properties of the `THdbc` class. Of course these two properties can also be specified as key-value pairs in the `THdbc.Attributes` property using the standard `UserName` and `Password` keys valid for all databases. Now we need to specify the ODBC driver to connect to:

```
Hdbc.Driver:= 'SQL Server';
```

The ODBC Driver Manager will pass all the above information to the specified ODBC driver. Then you just call the `THdbc.Connect` method to establish the DataSource-less connection.

## *Connection Pooling*

Establishing a connection through ODBC is not always instantaneous. This is because the ODBC Driver Manager first has to load the appropriate ODBC driver into memory before it can instruct the driver to establish the connection with the database. Then there is also the delay in establishing the connection (verifying of login information, etc.). This can be a problem when establishing multiple connections to a database (e.g. in a client/server environment requiring one connection per user on the server).

To bypass this problem ODBC provides the concept of connection pooling, whereby the ODBC Driver Manager keeps a pool of connections, and then when you try to establish a new connection, it first checks if an appropriate connection is not in the pool before a new connection is established. If the ODBC Driver Manager finds a suitable connection in the connection pool which matches a connect request, it will use this connection rather than creating a new connection, resulting in virtually instantaneous connection time. In this way a single connection can be shared by any number of users or applications. Connections are also kept in the connection pool for a long time after it's not in use any more, only dropping the connection after its "not-in-use" time in the pool has expired or when the application with the ODBC environment controlling the connection pool is closed. In other words connection pooling is also advantages in a client application where a connection to a database is established and dropped multiple times. It will reduce re-connection times drastically.

For example, say we want to create a server application with connection pooling enabled. We can enable connection pooling either by setting the `ConnectionPooling` property of the `THdbc` component or the `ConnectionPooling` property of the `GlobalHenv` component to `True`. Setting `ConnectionPooling` to `True` at either the `THenv` or `THdbc` level will affect all connection components used in the application. `ConnectionPooling` is actually an environment-level function, however it can be set a `THdbc` level also to allow setting of this property via the Object Inspector.

## *Transactioning*

Transactioning in ODBC is done at connection handle level. The connection component in ODBCExpress provides methods to start, end, commit and rollback transactions. By default, the connection component is in auto-commit mode. This means that all operations done at a DataSource using the child statement components of a connection component are automatically committed. Starting transaction mode takes the connection component out of auto-commit mode and starts a new transaction. Performing multiple operations in transaction mode is much faster than performing the operations out of transaction mode, since an auto-commit won't be done after each operation.

A new transaction is started after each call to `THdbc.StartTransact`, `THdbc.Commit` and `THdbc.RollBack`, while a transaction is ended by each call to `THdbc.Commit`, `THdbc.RollBack` and `THdbc.EndTransact`. Note that a transaction only starts at the first statement call made after `StartTransact`, `Commit` or `RollBack` was called. Also, calling `Commit`, `RollBack` or `EndTransact` without preceding it with transactional operations has no effect and won't cause any errors. Always match each `StartTransact` with an `EndTransact`, preferably as part of a try-finally statement, otherwise your application will end up in an invalid transaction state when an exception occurs. For example:

```
//start manual commit mode
Hdbc.StartTransact;

try

  try

    //perform database operations

    //commit transaction
    Hdbc1.Commit;

  except
    on EODBC do
      //rollback transaction when an ODBC exception occurs
      Hdbc.Rollback;
  end;

finally
  //end manual commit mode in finally block
  Hdbc.EndTransact;
end;
```

Because transactioning is done at connection component level, it means that the transactions done on different connection components within the same application are isolated from each other, as well as transactions done from different applications. Care must therefore be taken when using multiple connection components in the same application to avoid locking problems. However, different statement components on the same connection component aren't isolated from each other when in transaction mode, which means you can make use of multiple statement components within a single transaction to perform the operations you require at a DataSource. The type of transaction isolation done can be set using the `THdbc.IsolationLevel` property. It can be one of the following four values:

- ***Read Uncommited*** - Dirty reads, non-repeatable reads and phantom reads are possible.

- ***Read Comitted*** - Dirty reads are not possible. Non-repeatable reads and phantom reads are possible.

- ***Repeatable Read*** - Dirty reads and non-repeatable reads are not possible. Phantom reads are possible.

- ***Serializable*** - Transactions are serializable. Dirty reads, non-repeatable reads and phantom reads are not possible.

The following terms are used to define the isolation levels:

- ***Dirty Read*** - Transaction 1 changes a row. Transaction 2 reads the changed row before transaction 1 commits the change. If transaction 1 rolls back the change, transaction 2 will have read a row that is considered to have never existed.

- ***Non-repeatable Read*** - Transaction 1 reads a row. Transaction 2 updates or deletes that row and commits this change. If transaction 1 attempts to reread the row, it will receive different row values or discover that the row has been deleted.

- ***Phantom Read*** - Transaction 1 reads a set of rows that satisfy some search criteria. Transaction 2 inserts a row that matches the search criteria. If transaction 1 re-executes the statement that read the rows, it receives a different set of rows.

Of course, the transaction isolation level you choose influences the concurrency level between different transactions. A “read uncommitted” isolation level provides the most concurrency, while the concurrency reduces using “read-committed” through “repeatable read” until “serializable” which provides the least amount of concurrency.

To make transactioning a bit easier, make use of a small number of transactioned connection components within your application (preferably one if the type of application allows it) to avoid locking problems between these connections in your application. Also make use of a transaction isolation level which provides the highest level of concurrency, without compromising the integrity of your data, to provide less opportunity for locking problems.

# *SQL Operations*

Data at a DataSource is manipulated through ODBC via SQL operations. SQL operations form a main part of the ODBC client/server architecture and it is therefore important that you have a good knowledge of how to perform SQL operations. In fact, ODBC is completely SQL driven and doesn’t allow you to manipulate data in other forms. This is one of the main differences between ODBC and the BDE, and also accounts for a lot of the misunderstanding when attempting to move from the BDE to ODBCExpress.

The BDE does a lot of data manipulation on the front-end, and allows you to manipulate data in non-SQL-oriented fashions, whether using a SQL-oriented database or not. While this provides a lot of advantages for non-SQL-oriented databases, this also presents some performance problems for SQL-oriented databases. Because the Delphi database component architecture hides the obvious differences between SQL-oriented and non-SQL-oriented data manipulation, it makes it very difficult for the Delphi database developer to distinguish between the two depending on what type of database they’re using, and results in the development of poor client/server applications.

ODBC (and therefore ODBCExpress) on the other hand mainly use SQL operations to manipulate data, and will therefore not provide you with direct access to non-SQL databases functionality, however it provides all the SQL functionality necessary to get the best performance out of SQL databases. Both methods of manipulating data at a database have their merits depending on the type of database you’re using, and we will therefore attempt to highlight the differences between SQL and non-SQL operations throughout this white paper.

Data manipulation can basically be divided into two main sections:

- Manipulation without direct visual user involvement.

- Manipulation through direct visual user involvement.

ODBCExpress distinguishes between these two types of data manipulation by providing a main component (`THstmt`) which is geared towards non-visual or indirect visual development (with its own unique syntax), and a component (`TOEDataSet`) based on the previously mentioned component which is geared towards direct visual development (though this second component also allows you to do non-visual or indirect visual development using a more familiar BDE-like syntax).

The ODBCExpress `THstmt` statement component (another class defined in the OCL) encapsulates the ODBC statement handle, and is used to manipulate data at a DataSource. For BDE users, this component is equivalent to the `TQuery` BDE component, which also allows you to manipulate data at a DataSource given a SQL statement. The BDE `TTable` component on the other hand manipulates data at a DataSource in a non-SQL-oriented fashion, i.e. it allows you to manipulate not only the data in a specific table in a non-SQL-oriented manner, but also the table structure. While this component works very well for certain flatfile databases, it performs poorly against SQL databases and it's therefore better to use the `TQuery` component in such cases.

The ODBCExpress `TOEDataSet` statement component (a class defined in the ODSI) is derived from the virtual Delphi `TDataSet` component, which allows you to connect ODBCExpress with the standard Delphi data-aware controls, as well as data-aware controls (which conforms to the Open Database Architecture of Delphi) developed by third-party developers. This component is very similar to the BDE `TQuery` component, though the implementation is somewhat different.

General SQL operations include inserts, updates, deletes and selects. However SQL can also be used to perform any other function at the database, such as manipulation of databases, tables, views, indexes and stored procedures.

## *Data Manipulation using the THstmt*

The `THstmt` component provides all the operations necessary to perform SQL operations at a database. Let's take a typical example of building a result set. First we set the SQL statement to select all the students registered in a certain year:

```
Hstmt.SQL:= 'SELECT SNo, SName, SAge FROM Students WHERE SNo LIKE ?';
```

Now we can prepare the SQL statement. Preparing SQL statements aren't always necessary – in fact if you are only going to execute the SQL statement once, then it will be faster without preparing it (since the SQL statement will be parsed on the execute in this case), however if you're going to execute the SQL statement multiple times (as we're going to do), then preparing it once and then executing it multiple times will be faster (since you are avoiding parsing of the SQL statement each time you execute):

```
Hstmt.Prepare;
```

Now we must bind in the parameter indicated by the `?` in the SQL statement. The `?` indicates an un-named parameter. The `THstmt` class allows you to specify both named and un-named parameters. Let's change the above SQL statement with a named parameter:

```
Hstmt.SQL:= 'SELECT SNo, SName, SAge FROM Students WHERE SNo LIKE :SNum';
```

The named parameter starts with a `:` and can be followed by `A..Z`, `a..z`, `0..9`, the _ and @ characters. By the way, all colons `:` in the SQL statement will be passed as parameters, so in a case like the following you will get some problems:

```
Hstmt.SQL:= 'SELECT * FROM Files WHERE Path = ''C:\Program Files''';
```

To specify single colons in you SQL statement you must use a double colon `::`, as follows:

```
Hstmt.SQL:= 'SELECT * FROM Files WHERE Path = ''C::\Program Files''';
```

 

If you specify a named parameter, you can bind the parameter either by name or by number, for example the following are equivalent statements:

```
Hstmt.BindStringByName('SNum', ASNum);
Hstmt.BindString(1, ASNum);
```

By the way, this is also equivalent to:

```
Hstmt.BindString(Hstmt.ParamByName('SNum'), ASNum);
```

However if you've used an un-named parameter `?` in the SQL statement, then you can only bind the parameter by number:

```
Hstmt.BindString(1, ASNum);
```

Now let's bind our parameter. The `THstmt` component (unlike the `TQuery` and `TOEDataSet` components) requires you to bind in physical memory to a parameter, for example the following is valid because an integer has physical memory allocated up front:

```
var
  i: Integer;

Hstmt.BindInteger(3, MyInt);
```

However binding a string directly after it's declared is not valid:

```
var
  s: String;

Hstmt.BindString(3, s);
```

This is because strings in 32bit Delphi and C++Builder are dynamic and don't have memory allocated at the time they are being bound. However allocating memory to the string, binding it and then assigning a value to the string is also not valid:

```
Setlength(s, 100);  //max field length is 100

Hstmt.BindString(3, s);  //valid because memory allocated for string

s:= 'xyz';
```

This last statement is not valid since assigning a new value to a string might cause the string to be re-allocated, making the bound memory invalid. You will have to copy the value in the string:

```
StrPCopy(PChar(s), 'xyz');
```

If you only going to assign one value to the string, then of course the following strategy is the best - assign the value to the string before binding it:

```
s:= 'xyz';

Hstmt.BindString(3, s);  //valid because memory allocated for string
```

If you're going to execute the SQL statement multiple times with multiple values for the parameter, then you only need to bind the parameter once, assigning new values to the bound parameter each time you execute. In this case you will have to set the maximum length of the string up front and copy the values in the string before executing, as illustrated above. Or you can make use of a standard-length string structure such as the `NullString` type declared in the OCIH unit:

```
var
  s: NullString;

Hstmt.BindNullString(3, s);  //valid since the NullString type is not dynamic
                               and has memory allocated

s:= 'xyz';  //valid since no dynamic re-allocation of memory can occur.
```

Let's go back to our Students example and bind in the ASNum variable:

```
var
  ASNum: NullString;

Hstmt.BindNullStringByName('SNum', ASNum);
```

Now let's assign a value to ASNum and execute to return the result set of, say, all the students registered in 1998:

```
ASNum:= '98%';  //all student numbers starting with 98

Hstmt.Execute;
```

Now we can fetch the rows in the result set and for example add them to a TListBox:

```
ListBox.Clear;
while Hstmt.FetchNext do
  ListBox.Items.Add(Hstmt.ColString[1],
                    Hstmt.ColString[2],
                    IntToStr(Hstmt.ColInteger[3]));
```

Of course, we can also reference the columns by name, and retrieve the SAge integer column directly as a string:

```
ListBox.Clear;
while Hstmt.FetchNext do
ListBox.Items.Add(Hstmt.ColStringByName['SNo'],
                  Hstmt.ColStringByName['SName'],
                  Hstmt.ColStringByName['SAge']);
```

## *Data Manipulation using the TOEDataSet*

Now let's do the same using the TOEDataSet component (for BDE users this component has similar syntax to the TQuery component):

```
OEDataSet.SQL:= 'SELECT SNo, SName, SAge FROM Students WHERE SNo LIKE :SNum';
```

In the TOEDataSet component you can only use named parameters in the SQL statement, and not un-named ? parameters. This is because the TOEDataSet (and TQuery) builds parameter objects which need to be named. However you can still reference parameters by name or by number:

```
OEDataSet.ParamByName('SNum').AsString:= '98%';
OEDataSet.Params[0].AsString:= '98%';
```

Parameters in the TOEDataSet are zero-based, unlike the THstmt component in which they are one-based. Physical memory for a parameter is created and bound automatically, and a statement like the above will then automatically copy the value to the bound memory.

Now we need to open the dataset and retrieve all the rows in the result set:

```
OEDataSet.Open
```

 

```
ListBox.Clear;
while not OEDataSet.EOF do
  ListBox.Items.Add(OEDataSet.FieldByName('SNo').AsString,
                    OEDataSet.FieldByName('SName').AsString,
                    OEDataSet.FieldByName('SAge').AsString);
```

The advantage of Delphi 3 and C++Builder 3 over the previous versions of these products is the new Open Database Architecture, which allows third-party database middleware products such as ODBCExpress to make use of the Delphi data-aware control architecture. By deriving a descendant such as the `TOEDataSet` from the new virtual `TDataSet`, it gives such descendants the ability to make use of the standard Delphi and other 3rd-party data-aware controls. The `TOEDataSet` was therefore implemented for two main reasons:

- To give existing ODBCExpress users access to the standard Delphi and other 3rd-party data-aware controls and reporting tools which complies to the Open Database Architecture in Delphi 3 and C++Builder 3.

- To allow existing BDE and other database engine users to take advantage of ODBC and ODBCExpress performance and features, without losing the visual aspect of their applications.

Additionally the `TOEDataSet` provides `TQuery`-like properties and methods, to allow some level of code compatibility with existing applications, to ease the move from other database engines to ODBCExpress. The `TOEDataSet` also contains a `Hstmt` sub-property, which contains all the appropriate properties and methods of the ODBCExpress `THstmt` statement class relevant to the `TOEDataSet`.

# *Result Set Models*

One of the biggest differences between ODBCExpress and the BDE is the way in which they provide result set scrolling capabilities to your application. There are basically two models which provide you with result set scrolling:

- ***Cursor Model*** - Scrollable database cursors are used to provide scrolling through a database result set.

- ***Cache Model*** - The database result set is retrieved to a front-end cache using a non-scrollable cursor, and then scrolling is provided via a cursor kept on the cache.

The BDE makes use of the cache model only, while ODBCExpress provides both the cache model (via the ODBC Cursor Library) and also the cursor model. The full ODBC driver cursor support provided by ODBCExpress eliminates the need for a resource-heavy front-end cache as used by database engines such as the BDE. Cursor-support allows you to move backwards in your result set without having to cache the rows on the front-end. This means scrolling to the last row in your result set takes about the same time as scrolling to the first or next row in your result set. Also, since blobs are fetched from the database as they are needed, this means that you can have as many blobs as you want in your result set, no matter what the size of the blobs or how many rows you have in your result set. The cache model is fine for pretty small result sets with few or none blobs, however imagine building a 10 000+ row result set (which might contain blob columns) with a cache model - this means retrieving all the rows (including the blobs) to the front-end cache. The cursor model therefore allows you to work with very large result sets.

The cache model also has its advantages. Non-scrollable database cursors are generally much faster than scrollable cursors, so if you have a relatively small result set, the cache model will be a better choice. For example, when doing direct visual interaction with users, result sets tend to be smaller. Because the non-scrollable cursor used by the cache model is faster than scrollable cursors used by the cursor model, functionality such as Locates, Lookups and Filters should be much faster than with the cursor model. The BDE also takes advantage of the cache model to do complex matching (which in a SQL-oriented cursor model would generally be done as part of the WHERE-clause in your SQL statement). This functionality used by Locates, Lookups and Filters are therefore implemented in the BDE itself and does not form part of the `TDataSet` in any way, in other words it's not handled by the Delphi portion of the BDE code. Because ODBCExpress make use of the ODBC Cursor Library to implement a cache model, it cannot access the cache to do complex row matching and therefore has to simplify the type of row

matching allowed. The same goes for the cursor model provided by ODBCExpress. Locates, Lookups and Filters are therefore implemented by ODBCExpress and not by ODBC, and will work best with a fast cursor, as used by the ODBC Cursor Library.

A cursor model provides a lot of advantages, especially when it comes to performance. To quote from a Microsoft white paper on using server cursors:

"There are several advantages to using server cursors:

- ***Performance*** - If you are going to access a small fraction of the data in the cursor (typical of many browsing applications), using server based cursors will give you an optimal performance because only required data (and not the entire results set) is sent over the network.

- ***Additional cursor types*** - Keyset and dynamic cursor types are only available if you use server cursors. In addition, the cursor library only supports forward only with read-only concurrency and static with read-only and optimistic concurrency. With server based cursors, you can use the full range of concurrency values with the different cursor types.

- ***Memory usage*** - When using server based cursors, the client does not need to cache large amounts of data or maintain information about the cursor position; the server provides that functionality.

- ***Multiple open cursors*** - When using server based cursors, the connection between the client and the server does not remain busy between cursor operations. This allows you to have multiple cursor hstmts active at the same time"

So you can see the cursor model gives you better performance and more control over concurrency and resource usage, which is pretty obvious if you ponder about the differences between a cache and cursor model. This is the fundamental difference between ODBCExpress and the BDE.

## *Cursor Types*

Let's take a look at the different cursors types ODBCExpress provides for the cursor model:

- ***Forward-Only*** - Because this is a non-scrollable cursor, the cursor only moves in a forward direction. It is generally much faster than the scrollable cursors listed below. This cursor does not build a result set at the database, therefore rows not fetched yet will reflect any changes made to them by the time they are fetched.

- ***Static*** - This is a scrollable cursor and the data and rows which makes up the result set is static. In other words all rows modified, added or deleted at the database by other users do not influence the result set. To achieve this effect a result set is built at the database (probably by duplicating the rows in a temporary database), making this the slowest of the scrollable cursors.

- ***Keyset-Driven*** - The data in the result set is dynamic, but the rows which makes up the result set is static. In other words all rows in the result set modified by other users are reflected by the result set, however the rows that make up the result set are not influenced by inserts or deletes made by other users. To achieve this only a result set of key values which identify all the rows in the result set is built at the database, making this a very fast scrollable cursor.

- ***Dynamic*** - The data and rows which makes up the result set is dynamic. In other words all rows modified, added or deleted at the database by other users are reflected by the result set. As the forward-only cursor this cursor does not built a result set at the database, but only keeps a scrollable cursor on the data in the result set, also making this a very fast (probably the fastest) scrollable cursor.

Since the rows which make up the result sets in Static and Keyset-Driven cursors remains static, it's possible to retrieve bookmark values for these cursors, since bookmarks are generally based on position in a result set and remains valid for the duration of the result set only. Forward-only and Dynamic cursors generally aren't able to

return bookmark values for rows (since the rows which make up the result set does not remain static), unless bookmarks are based on positions within the database tables (which is a rare occurrence). In this last case the bookmarks will probably remain valid even after the result sets are closed.

Where possible forward-only cursors should be used, since these cursors will provide you with the best retrieval performance. However if scrolling is required (normally during direct user-interaction), then one of the scrollable cursors can be used in the following order:

- ***Dynamic***, if bookmarking isn't required and changes made by other users has to be fully reflected. Also this is a very fast scrollable cursor.

- ***Keyset-Driven***, if bookmarking is required and changes made to the values of rows must be reflected. This is also a very fast scrollable cursor.

- ***Static***, if bookmarking is required, as well as complete isolation from changes made by other users.

In the `THstmt` component any of these cursor types can be used. When using the `TOEDataSet` for non-visual purposes, and no scrolling is required, then the forward-only cursor is the best option. However if scrolling in the `TOEDataSet` is required (for visual or non-visual purposes), then any scrollable cursor which provides bookmarks can be used, since the `TOEDataSet` requires a bookmarked cursor to provide scrolling (where as the `THstmt` doesn't require a bookmarked cursor to provide scrolling). This is because the virtual `TDataSet` needs some sort of bookmarking for its scrolling functionality to work correctly. Because the `TOEDataSet` requires a bookmarked cursor for scrolling, this generally rules out the Dynamic scrollable cursor. However some databases (e.g. MS Access) does provide bookmark values for the Dynamic cursor, which means you can use the Dynamic cursor together with the `TOEDataSet` and these databases.

So which is the best scrollable cursor to use with the `TOEDataSet`? This depends on the type of functionality you are implementing. For example say we want to populate a `TDBGrid` using the `TOEDataSet`. Suppose we use a Dynamic cursor which supports bookmarks. This will work fine in a single-user environment, however what will happen in a muti-user environment when another user deletes one of the rows which are visible in the grid? When you scroll back to that row (which were just deleted), the row won't exist in the result set anymore (because the rows which make up the result set of a Dynamic cursor aren't static), so the grid in that position will be filled with the values of the next or previous row in the grid (depending on whether you are scrolling down or up respectively). This will result in a duplicate-row effect in which the current and next or previous row in the grid will have the same values (since they indicate the same row in the result set). So in a multi-user environment it will be better to use a Static or Keyset-Driven cursor with the `TDBGrid`, which will guarantee that the rows which make up the result set remains static. Of course if you're only using single-row controls (e.g. `TDBEdit` controls) which only display one row at a time the duplicate-row effect does not occur. So you can see that either a Static or Keyset-Driven cursor can safely be used with the `TDBGrid`.

Now let's take a look at the cursor types that can be used with the cache model in ODBCExpress. Because the cache model retrieves the rows in the result set to a front-end cache (on demand of course), it won't be influenced by any changes made by other users for the duration of the result set – the rows and values of the rows that makes up the result set remains static. You thereby have the options of forward-only or static-like cursors only in both the ODBCExpress and BDE cache models.

The cache model in ODBCExpress makes use of the ODBC Cursor Library. To enable the cache model in ODBCExpress, you have to set the CursorLib property of the `THdbc` connection component. Also, if the ODBC driver you're using doesn't support scrollable cursors, you can make use of the ODBC Cursor Library. The cursor library not only provides you with a scrollable cursor for your result set, but also enables the ODBC driver to perform positional operations (which we'll discuss later) if your ODBC driver doesn't support it directly. Let's take a look at the possible values of the `THdbc.CursorLib` property:

- ***Use Driver*** - Use the cursor model and the ODBC driver's scrolling capabilities.

- ***Use ODBC*** - Use the ODBC Cursor Library to enable the cache model or provide scrolling capabilities to your ODBC driver.

- ***Use If Needed*** - Use the ODBC Cursor Library only if the ODBC driver doesn't support scrollable cursors. Though this is a useful option, it's better practice to make the decision yourselve.

## *Concurrency Types*

There are two distinct ways in which you can modify data at a database in a SQL-oriented environment. The most common way is by specifying INSERT, UPDATE or DELETE SQL statements and execute it to insert, update or delete rows. The other way is by building a result set using a SELECT SQL statement, and then positionally insert, update or delete rows in the result set. Whether you can perform this last set of positional operations on the result set depends on whether the result set is modifiable. In the terminology of BDE users, whether you can perform deletes in the dataset or post inserts and updates in the dataset depends on whether you have a "live" dataset or not. The ODBC cursor model used by ODBCExpress allows you to set the following concurrency types, which also controls the modifiability of the result set:

- ***Read-Only*** - The cursor is read-only. No positional updates or deletes are allowed (however you can still do updates and deletes via normal SQL). This option has a high concurrency level.

- ***Locking*** - The cursor uses the lowest level of locking sufficient to ensure that rows can be updated or deleted. This option results in the lowest concurrency levels of the four types of concurrency, however it ensures that a row can be updated or deleted.

- ***Optimistic Concurrency using Row Versions*** - The cursor uses optimistic concurrency control, comparing row versions to detect changes. The updates or deletes are performed only if the row has not changed since the last read. It provides a high level of concurrency, however there is no guarantee that a specific row can be updated or deleted.

- ***Optimistic Concurrency using Row Values*** - The cursor uses optimistic concurrency control, comparing row values to detect changes. The updates or deletes are performed only if the row have not changed since the last read. It provides a high level of concurrency, however there is no guarantee that a specific row can be updated or deleted.

When using the cache model, concurrency can only be controlled by comparing row values before updates or deletes are done. You thereby have the options of read-only or row-values-like concurrency types only in both the ODBCExpress and BDE cache models.

## *Positional Operations*

Let's take a closer look at the positional way of updating data at a database. Because this can only be done via a result set, we first have to build the result set on which we wish to perform these operations. We can now perform an insert, update or delete positional operation in one of the following 3 ways:

- Using no SQL

- Using automatically generated SQL

- Using manually constructed SQL

These three options form the basis of ODBCExpress' so-called SQL Generation functionality for positional operations. When doing a positional insert, update or delete, the success of the operation depends on if the ODBC driver supports the operation and, as discussed in the previous section, if the result set is modifiable. These two issues can be are addressed by using the correct level of SQL Generation (provided of course that the user has the appropriate rights to modify the data in the result set). Four levels of SQL Generation are provided by ODBCExpress:

- ***Level 1*** - SQL-less operations using ODBC functions are done to perform inserts, updates and deletes in the result set. This requires the least effort and control, but requires a concurrency type which allows result set modification, and also uses Level 2 ODBC functionality (which aren't always supported by all ODBC drivers).

- ***Level 2*** - Automatically generated SQL, together with result set cursor positions, are used to perform inserts, updates and deletes in the result set. Complete SQL statements are generated to perform each operation, making use of the result set cursor to determine the row affected by the operation. It might be necessary to supply the name of the table which will be affected, but only if the ODBC driver is not able to detect it automatically. This level also requires a concurrency type which allows result set modification.

- ***Level 3*** - Automatically generated SQL, together with supplied primary keys (and affected table if necessary) , are used to perform inserts, updates and deletes in the result set. Complete SQL statements are also generated to perform each operation, but in this case the supplied primary key columns are used to determine the rows affected by the operation. The biggest advantage of this level is that you don't need a concurrency type allowing result set modification to perform this operation. This is essentially not a positional operation and as said in the previous section, concurrency types don't affect non-positional operations.

- ***Level 4*** - Manual SQL construction via events are used to perform inserts, updates and deletes in the result set. This is done by supplying your own SQL statements to perform the operations using the OnInsert, OnUpdate or OnDelete events of the THstmt and TOEDataSet components. This is also a non-positional operation and therefore doesn't require a modifiable result set.

Levels 2 and 3 can be compared to the functionality performed by the BDE `TUpdateSQL` component, which of course eliminates the need for such a component in ODBCExpress.

As explained levels 3 and 4 can be used to modify potentially non-modifiable result sets. A result set is non-modifiable when the concurrency type of the result set is read-only, or if the result set was returned by a query which makes it non-modifiable, such as a join. Using these two levels will allow you to bypass the non-modifiable state of the result set, allowing the user to do non-positional inserts, updates and deletes (provided of course that the user has the appropriate rights to modify the data at the DataSource).

Positional inserts, updates and deletes are performed using the `DoInsert`, `DoUpdate` and `DoDelete` methods of the `THstmt` statement component, or the `Insert`, `Edit`, `Delete` and `Post` methods of the `TOEDataSet` component (or `TQuery` component for BDE users).

So how do you specify in ODBCExpress which level of SQL Generation must be used to perform the desired positional operation? Let's take a closer look at that:

- Level 4 SQL Generation can override any of the other levels by implementing the appropriate operation inside the `OnInsert`, `OnUpdate` or `OnDelete` events, and then returning `False` from each event to avoid performing any of the first 3 levels. As an aside these events can therefore also be used to confirm an insert, update or delete operation, by returning the appropriate boolean value from each event.

- When Level 4 SQL Generation isn't done, by default Level 1, 2 or 3 is done, depending on the first level supported by the ODBC driver used. ODBCExpress will try to detect which of these 3 levels can be performed by the ODBC driver. If Level 1 SQL Generation is not supported by the driver, Level 2 is attempted and if Level 2 is also not supported by the driver, then Level 3 is done. Level 3 SQL Generation will work for any ODBC driver, no matter what the API conformance level of the ODBC driver is.

- Level 1 SQL Generation can be completely bypassed by setting the `SkipByPosition` property of the `THstmt` or `TOEDataSet` component to `True`. Then only Level 2 or Level 3 will be attempted when Level 4 isn't done.

- Level 2 SQL Generation can be completely bypassed by setting the `SkipByCursor` property of the `THstmt` or `TOEDataSet` component to `True`. Then only Level 1 or Level 3 will be attempted when Level 4 isn't done.

- Level 1 and Level 2 SQL Generation can be completely bypassed by setting both the `SkipByPosition` and `SkipByCursor` properties of the `THstmt` or `TOEDataSet` component to `True`. Then only Level 3 will be attempted when Level 4 isn't done.

Now let's quickly take a look at what happens to the result set when you positionally insert, update or delete a row in a static-type result set as normally used with the `TOEDataSet` component. When you insert or delete a row from the result set, the row is not physically added to or removed from the result set, because the rows which make up the result set is static. Visually this means that rows that you insert into the result set will disappear from view and rows that you delete from the result set will not disappear from view. Also, when updating a row using a static result set in which the values in the result set also remain static, visually this means scrolling off the row and then back onto the row, the row would display its original values.

The required behavior for `TDataSet` descendants is however that inserted rows must always stay for the duration of the open dataset at the position they were inserted, deleted rows must disappear from the open dataset and modified rows must reflect the modifications for the duration of the dataset. Now the last two cases are pretty easy to implement in the `TOEDataSet`, ODBCExpress' descendant of the `TDataSet`. The `TOEDataSet` just keeps track of the deleted and modified rows and adjust the visual display of the dataset accordingly. However, because in a SQL-oriented static-type result set inserted rows have no physical position and existing rows in the result set also has no physical positions, there is no way to manually keep track of inserted rows, causing the so-called "disappearing inserts" effect with the `TOEDataSet`. The only way this problem can be bypassed is to keep track of the primary keys of the inserted row, close and re-open the result set, and then position to the inserted row which might now form part of the new result set. Of course if the inserted row doesn't match the search-criteria of the new result set, then this won't be possible. The `TOEDataSet` has a `LocateInsert` boolean property which does just this. If set to `True`, it uses the target primary keys to extract the primary keys of the inserted row, then closes and re-opens the dataset and locates the inserted row doing a multiple-field `Locate` on the dataset with the extracted primary keys.

# *Blob Handling*

Storing and retrieving blob fields at a DataSource with ODBCExpress is just as easy as manipulating any other field at the DataSource. Fields in a database which can hold information larger than 256 bytes are generally BLObs (Binary Large Objects). You usually get two types of blobs:

- ***Text blobs*** - consist of a string of readable characters which can't be fitted into a normal 256 character-limited string field.

- ***Binary blobs*** - consist of a stream of unreadable characters which is to be stored in binary format at the database.

The ODBCExpress statement classes also provide various properties and methods to allow you to customize blob handling for a specific database, if necessary.

# *Bulk Operations*

The `THstmt` statement component allows you to perform operations not possible through the Open Database Architecture of Delphi and C++Builder. It allows you to perform bulk fetches (retrieving multiple rows with each fetch) and bulk inserts (inserting multiple rows with one execute) to help you speed up data-intensive operations and reduce network traffic.

## *Threading*

Both ODBC and ODBCExpress are thread safe. Also, because ODBC is thread safe at environment level for all ODBC drivers, only one THenv environment component (GlobalHenv) is needed when using multiple threads. Whether you can also split a THdbc or THstmt component across threads depends on whether the ODBC driver you are using is thread safe at these levels. ODBCExpress also provides built-in functionality to allow cancelable queries with both the THstmt and TOEDataSet component, all which can be done from a single thread if necessary.

## *Stored Procedures*

ODBCExpress provides you with integrated stored procedure handling on the statement components – it's as easy and works the same as with any other query performed against these components.

## *Other Components*

ODBCExpress contains a number of components which helps you with database schema manipulation (`TOESchema`), database catalog querying (`TOECatalog`), DataSource querying and manipulation (`TOEAdministrator`) and ODBC and ODBC driver installation and setup (`TOESetup`).

## *Non-Visual Development*

ODBCExpress also excels in non-visual environments such as web development and server-side applications, because of its distinct division between non-visual and visual code, providing the full functionality needed in both portions of the code to perform any database operation.

## *Future of ODBCExpress*

Currently ODBCExpress is tied to the Borland Development Languages which support the Visual Component Library (VCL), namely Delphi and C++Builder. However because of Delphi's unprecedented support for the Component Object Model (COM), the platform and language independent binary standard that defines how objects are created and destroyed and how they interact with each other, the next step will be to design a set of automation objects and ActiveX controls around the ODBCExpress architecture. This will allow ODBCExpress to be used in other development languages such as Microsoft's Visual C++ and Visual Basic, but especially from web servers and Java applications.

Of course because the COM version will be completely based on the current Delphi and C++Builder versions, it means that continued development on these current versions will form a very important part of the process. In fact the COM version will justify much more resources and support behind ODBCExpress.