

I Seminarium PLOUG
Warszawa
Marzec 2001

We wnętrzu dbExpress

Nowa, wieloplatformowa warstwa dostępu do danych Borlanda

Ramesh Theivendran

Borland R&D

W projekcie Kylix wprowadzono technologię dbExpress. Znajdzie się ona również w przyszłych wersjach Delphi i C++Buildera dla Windows.

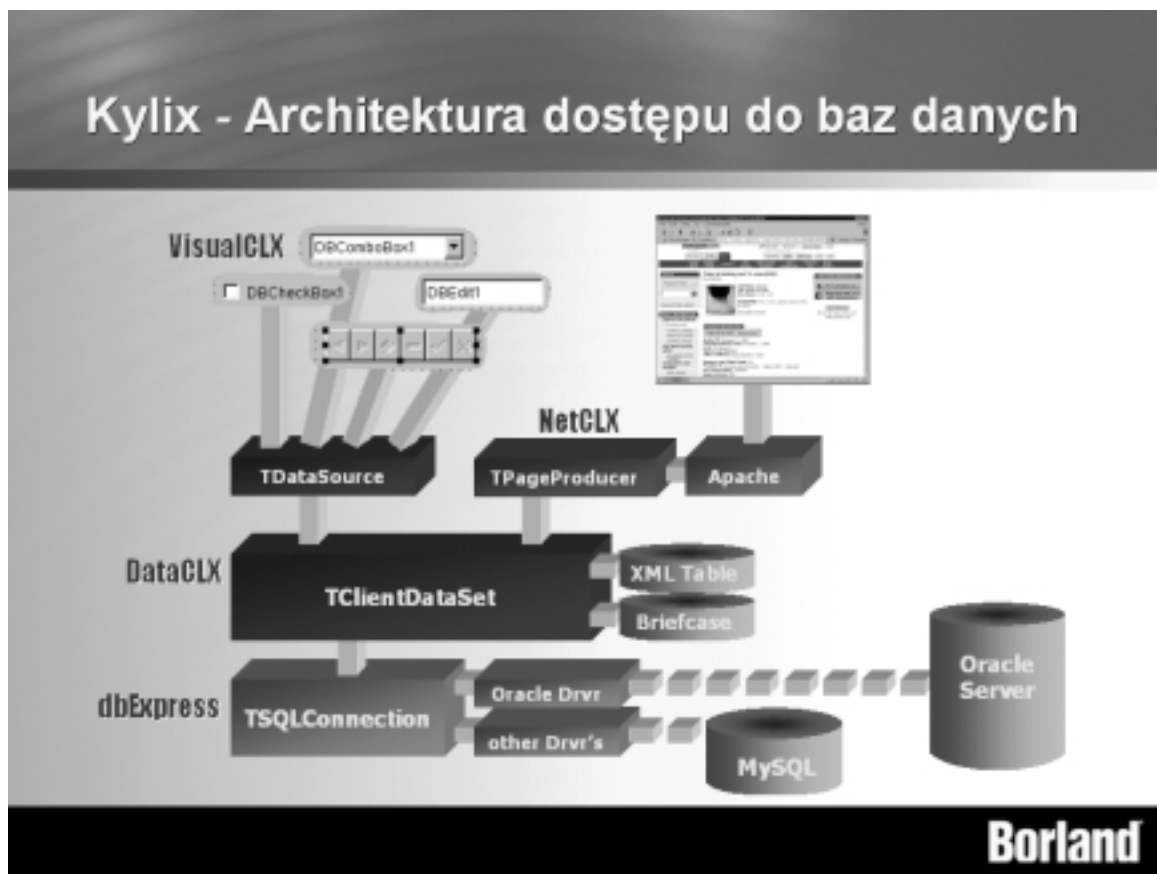
Ramesh Theivendran

Wprowadzenie

Przechowywanie i dostęp do danych odgrywają główną rolę w dzisiejszym przemyśle IT. Trudno byłoby znaleźć poważną aplikację dla biznesu, która nie korzystałaby z bazy danych. Kluczowym elementem aplikacji bazodanowych jest wydajność.

W przeszłości, takie standardy jak ODBC i JDBC, zapewniły niezależną od bazy danych warstwę dostępu. Dzięki wspólnemu interfejsowi, specyficzna funkcjonalność ważna dla określonej bazy danych, zostaje zepchnięta w cień. Jednocześnie, próby uwzględnienia cech rozmaitych baz danych, powodowały, że warstwa dostępu do danych stawała się coraz bardziej złożona i obszerniejsza.

dbExpress to wieloplatformowy, niezależny od bazy danych i rozszerzalny interfejs zapewniający zestaw metod do dynamicznego przetwarzania SQL. dbExpress będzie pełnił rolę warstwy dostępu do danych następnej generacji dla Kylix, Delphi i C++Buildera. Głównym celem tego artykułu jest wyjaśnienie działania nowego interfejsu dostępu do danych, omówienia różnic względem istniejących standardów oraz tego, jak zaprojektować sterownik bazodanowy dla swojej ulubionej bazy danych.



dbExpress kontra BDE

Technologia dbExpress, podobnie jak BDE (Borland Database Engine), jest w stanie przetwarzać zapytania i składowane procedury. BDE jest bogatszym funkcjonalnie, większym klientem, natomiast technologia dbExpress jest prostsza i łatwiejsza w implementacji i najlepiej pasuje do modelu „provider and resolver” MIDAS-a.

Aplikacje intensywnie wykorzystujące bazy danych powinny dostawać się do danych przy użyciu zapytań i procedur składowanych. Otwieranie tablic poprzez BDE nie tylko konsumuje zasoby klienta, ale również blokuje zasoby serwera. dbExpress eliminuje koncepcję otwierania tablic, a klientom zaleca się stosowanie zapytań SQL z większą selektywnością dla zoptymalizowanego dostępu do danych.

W przeciwieństwie do BDE, dbExpress zwraca jedynie kursory jednokierunkowe i dlatego nie wykonuje buforowania. Do buforowania, przewijania, indeksowania i filtrowania zbioru wyników można wykorzystać lokalne zbiory danych MIDAS ClientDataset.

BDE buforuje metadane, aby były gotowe do wykorzystania przez przyszłe żądania. dbExpress nie buforuje ich, a interfejs dostępu w czasie projektowania jest zaimplementowany przy pomocy podstawowego interfejsu dostępu do danych.

Wydajność współdziałania BDE z bazami danych SQL jest pochodną wewnętrznego generowania zapytań do poruszania się po zbiorach danych, dostępu do danych BLOB i odczytywania metadanych. dbExpress wykonuje tylko zapytania zażądane przez użytkownika, dzięki czemu zmniejsza czas dostępu do bazy danych dzięki nie wprowadzaniu żadnych dodatkowych zapytań.

dbExpress wewnętrznie zarządza buforem rekordów i dostarcza klientom poszczególnych wartości pól. BDE wykorzystuje dla kontrastu bufor rekordów zaalokowany przez klienta, co może być źródłem błędów, ponieważ klient może przekazać bufor wadliwy lub niewystarczających rozmiarów.

Poza innymi różnicami względem BDE, dbExpress nie zawiera koncepcji żywych zapytań, buforowanych uaktualnień, obsługi tworzenia schematów, przetwarzania zapytań heterogenicznych, operacji „batchmove” itp.

dbExpress rozwiązuje również następujące problemy, które aktualnie trapią BDE:

- złożoność konfigurowania i wdrażania BDE oraz dostosowywanie do nowych źródeł danych;
- narzuty w czasie wykonywania programu pochodzące od buforowania danych BLOB, metadanych i ładowania konfiguracji BDE
- wydajność

Cechy sterowników bazodanowych

Różne cechy istniejących standardów łączności z bazami danych, takich jak BDE, ODBC i JDBC, można z grubsza pogrupować w trzy główne obszary: dostęp do danych, pobieranie metadanych i tworzenie schematów.

Najbardziej podstawową funkcjonalnością zapewnianą przez sterownik bazodanowy jest interfejs do przetwarzania dynamicznych zapytań SQL. Dynamiczne zapytania SQL umożliwiają aplikacji przetwarzanie dowolnych, poprawnych wyrażeń SQL w czasie wykonywania aplikacji. Bi-

bioteki klienckie większości producentów baz danych, takie jak OCI lub PRO*C (Dynamic Method 4) dla ORACLE, CLI dla BD2 i innych źródeł danych ODBC, CTLIB dla Sybase i E/SQL dla Informixa, zapewniają interfejs do przetwarzania dynamicznego kodu SQL.

Kiedy podstawowe przetwarzanie dynamicznego SQL jest już zbudowane, łatwo na nim później nadbudować odczytywanie metadanych i tworzenie schematów. Teraz przejdziemy do podstawowych elementów przetwarzania dynamicznego SQL.

Inicjalizowanie środowiska

Klienci pochodzący od większości producentów baz danych muszą najpierw zainicjalizować różne „uchwyty” zanim będą mogli połączyć się z serwerem bazy danych. Zwykle inicjalizacje te wiążą się zaalokowaniem zasobów klienta, takich jak SQLDA i SQLCA do komunikacji „end-to-end” z serwerem. X/Open SQL CAE wymaga zainicjalizowania uchwytu środowiskowego i uchwytu połączenia przez ustanowienie połączenia bazodanowego.

```
SQLHANDLE henv, hdbc;
SQLRETURN rc;
/* allocate an environment handle */
rc = SQLAllocHandle( SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv );
/* allocate a connection handle */
if ( SQLAllocHandle( SQL_HANDLE_DBC, henv, hdbc ) != SQL_SUCCESS )
{
    printf( "ERROR while allocating a connection handle-----n" );
    return SQL_ERROR;
}
```

Łączenie się z serwerem bazy danych

Po zaalokowaniu wymaganych uchwytów, można ustanowić połączenie z bazą danych podając serwer, nazwę użytkownika i hasło. Przy pomocy uchwytu połączenia można również ustawić lub pobrać wartości pewnych właściwości przed i po uzyskaniu połączenia, takich jak „autocommit on/off”, „connection time out” (właściwości ustawiane przed połączeniem) oraz informacja o wersji bazy danych (właściwość pobierana po ustanowieniu połączenia).

```
/* Set AUTOCOMMIT OFF */
rc = SQLSetConnectAttr( hdbc, SQL_ATTR_AUTOCOMMIT, ( void * ) SQL_AUTOCOMMIT_OFF, SQL_NTS );
if ( SQLConnect( hdbc, Server, SQL_NTS, UserName, SQL_NTS, Password, SQL_NTS ) != SQL_SUCCESS )
{
    printf( "ERROR while connecting to %s -----n", Server );
    return SQL_ERROR;
}
```

Inicjalizacja uchwytów wyrażeń

Przed przetworzeniem jakiegokolwiek wyrażenia SQL należy zaalokować dla połączenia uchwyt wyrażenia. Większość baz danych umożliwia posiadanie więcej niż jednego aktywnego wyrażenia dla jednego połączenia. Na serwerach, które tego nie umożliwiają, do jednoczesnego przetwarzania wielu wyrażeń SQL należy ustanowić oddzielne połączenia. Przy pomocy uchwytu wyrażenia można również ustawiać i odczytywać właściwości poziomu wyrażenia.

```
#define ROWSET_SIZE 20
SQLHANDLE hstmt;
rc = SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt );
/* Set BLOCK FETCH SIZE */
rc = SQLSetStmtAttr( hstmt, SQL_ATTR_ROW_ARRAY_SIZE, (SQLPOINTER) ROWSET_SIZE, 0);
```

Przygotowywanie wyrażenia SQL

Przygotowywanie wyrażenia SQL sprawdza jego poprawność i tworzy plan wykonania zapytania na serwerze. Kiedy wyrażenie SQL jest przygotowane, większość baz danych umożliwia jego wielokrotne wykonanie poprzez przekazywanie mu w czasie działania programu różnych parametrów. Po pomyślnym przygotowaniu wyrażenia SQL, klienci niektórych baz danych dostarczają informacji o rodzaju wyrażenia (INSERT, DELETE, UPDATE lub SELECT), ilości znaczników parametrów itp.

```
if ( SQLPrepare(hstmt, szSQL, SQL_NTS) != SQL_SUCCESS )
{
    printf( "ERROR while preparing a SQL statement -----n" );
    return SQL_ERROR;
}
```

Przekazywanie parametrów w czasie wykonywania programu

Parametry czasu wykonywania programu mogą być określane przez wyspecyfikowanie określonych znaczników w wyrażeniu SQL. Zależnie od bazy danych, z którą się komunikujemy, może być jeden lub więcej sposobów specyfikowania znaczników parametrów, jak wiązanie po nazwie lub wiązanie po numerze. Po udanym przygotowaniu sparametryzowanego wyrażenia SQL, należy dowiązać bufor parametrów i przed wykonaniem wyrażenia dostarczyć wartości parametrów lub ustawić znaczniki na NULL.

```
SQLCHAR insert_data[21] ;
SQLINTEGER insert_data_ind ;
for ( iPos = 1; iPos <= noOfParams; iPos ++ )
{
    ...
    ...
    switch ( dataType )
    {
        case SQL_CHAR:
            /* Bind data to parameter marker */
            rc = SQLBindParameter( hstmt, iPos, SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR, 20, 0,
                insert_data, 21, &insert_data_ind );
            break;
            ...
            ...
    }
}
```

Wykonywanie wyrażenia SQL

Wyrażenia SQL nie zawsze muszą być przygotowywane przed wykonaniem. Istnieją interfejsy umożliwiające bezpośrednie wykonanie wyrażenia SQL. Przy bezpośrednim wykonywaniu SQL, wszystkie zasoby klienta i serwera są zwalniane natychmiast po wykonaniu, a wszystkie kursory zbioru wynikowego są ściągane. Stąd, jeżeli aplikacja wymaga wielokrotnego wykonywania tego samego kodu SQL, najlepszym rozwiązaniem jest jednokrotne przygotowanie wyrażenia SQL, a następnie wykonywanie go wiele razy.

```
if ( SQLExecute( hstmt ) != SQL_SUCCESS )
{
    printf( "ERROR while executing a SQL statement -----n" );
    return SQL_ERROR;
}
rc = SQLExecDirect( hstmt, szSQL, SQL_NTS );
```

Przywiązywanie buforu rekordu

Po pomyślnym wykonaniu wyrażenia SQL, jeżeli jest to zapytanie SELECT, wtedy możemy odczytać metadane opisujące kolumny zbioru wynikowego. Dla zapytań INSERT, DELETE lub UPDATE możemy otrzymać informację o ilości zmodyfikowanych wierszy na serwerze. Jeżeli wiemy, że istnieje zbiór wynikowy, to musimy zaalokować bufor rekordów, który będzie wykorzystany do pobrania rekordów. Rozmiar bufora można obliczyć poprzez długości wszystkich kolumn zbioru wynikowego plus czterobajtowy znacznik NULL dla każdej kolumny. Jeśli na przykład w zbiorze wynikowym mamy dwie kolumny CHAR(20), wówczas rozmiar bufora rekordy będzie równy $(20 + 1 + 4) * 2 = 50$ bajtów. Zależnie od typu danych może zajść konieczność zapewnienie dodatkowego bufora na terminator NULL, uchwyty BLOB-ów itp. Dokładne wymagania rozmiarów bufora dla określonego typu danych można znaleźć w dokumentacji biblioteki klienckiej producenta bazy danych.

```
SQLSMALLINT noResultCols;
SQLINTEGER noofRowsAffected;
SQLNumResultCols ( hstmt , &noResultCols);
if ( noResultCols )
{
    /* Resultset is available */
    SQLCHAR colname[32] ;
    SQLSMALLINT coltype ;
    SQLSMALLINT colnamelen ;
    SQLSMALLINT nullable ;
    SQLINTEGER collen;
    SQLSMALLINT scale ;
    SQLINTEGER displaysize ;
    for ( iCol = 1; iCol<= noResultCols; iCol++ )
    {
        /* Describe each column in the resultset */
        rc = SQLDescribeCol( hstmt, ( SQLSMALLINT ) iCol, colname, sizeof(colname),
            colnamelen, &coltype, &collen, &scale, &nullable ) ;
        /* get display length for column */
        rc = SQLColAttribute( hstmt, ( SQLSMALLINT ) iCol, SQL_DESC_DISPLAY_SIZE, NULL,
            0, NULL, &displaysize ) ;
    }
}
else
    /* No resultset, should be a INSERT, UPDATE, DELETE or a DDL statement */
    rc = SQLRowCount ( hstmt , &noofRowsAffected );
```

Pobieranie rekordów

Po przygotowaniu buforu rekordu możemy przywiązać bufor do pobierania danych i wskaźników NULL poprzez iterowanie po wszystkich kolumnach zbioru wynikowego. Większość dostawców baz danych zapewnia więcej niż jeden sposób przywiązywania buforów, jak wiązanie według pozycji lub wiązanie według nazwy. Po zapewnieniu bufora dla wszystkich kolumn możemy wczytać rekord do bufora wywołując odpowiednie funkcje pobierające. Po wczytaniu rekordu należy sprawdzić wszystkie wskaźniki NULL kolumn, dla określenia czy w buforze znajdują się poprawne dane. Jeśli poprawne dane zostaną znalezione, kopiujemy dane z bufora rekordu do bufora zapewnionego przez aplikację. Kolejne operacje pobierania odczytują rekordy jeden po drugim, aż do osiągnięcia końca pliku (EOF).

```
for ( iCol = 1; iCol <= noOfColumns; iCol ++ )
{
    ...
    ...
    switch ( dataType )
    {
        case SQL_CHAR:
            /* Bind buffer for resultset column */
```

```

    rc = SQLBindCol(hstmt, iCol, SQL_C_CHAR, (SQLPOINTER) data, 15, indl);
    break;
    ...
    ...
}
}
while ((rc = SQLFetch(hstmt)) == SQL_SUCCESS)
{
    /* copy record to the client buffer */
    ...
    ...
}
if (rc == SQL_NO_DATA_FOUND)
{
    /* copy the last record to the client buffer */
    ...
    ...
    rc = SQL_SUCCESS;
}

```

Zwalnianie uchwytów i rozłączanie

W przypadku niektórych serwerów w wyniku wykonania składowanej procedury może zostać zwrócony więcej niż jeden kursor. Dlatego po pobraniu pierwszego kursora, można kontynuować pobieranie kolejnych kursorów, aż do osiągnięcia EOF. Po pobraniu wszystkich kursorów, jeżeli uchwyt wyrażenia SQL nie są już potrzebne do kolejnych wywołań, możemy zwolnić uchwyt wyrażenia, odłączyć się od bazy danych, zwolnić połączenie i wszystkie uchwytów.

```

rc = SQLFreeHandle( SQL_HANDLE_STMT, hstmt );
rc = SQLEndTran( SQL_HANDLE_DBC, hdbc, SQL_COMMIT );
/* Disconnect for the database */
rc = SQLDisconnect( hdbc );
rc = SQLFreeHandle( SQL_HANDLE_DBC, hdbc );
rc = SQLFreeHandle( SQL_HANDLE_DBC, henv );

```

Abstrakcja dbExpress

Cztery podstawowe klasy w dbExpress to `SQLDriver`, `SQLConnection`, `SQLCommand` i `SQLCursor`. Interfejs dostępu do metadanych, `SQLMetaData`, został zbudowany w oparciu o klasy podstawowe. Wszystkie klasy posiadają metody do ustawiania i pobierania właściwości czasu wykonania programu, z których mogą korzystać klienci w celu wykorzystania specyficznych cech bazy danych lub dla precyzyjnego dostrojenia dostępu do danych.

SQLDriver

Klasa `SQLDriver` wykonuje specyficzne dla danej bazy danych operacje inicjalizacyjne, takie jak załadowanie klienta bazy danych, zainicjalizowanie środowiska, zaalokowanie niezbędnych uchwytów i pobranie obiektu `SQLConnection`.

```

abstract COMINTF SQLDriver
{
    virtual INT32 GCCSTDC QueryInterface( GUID riid, ppVOID ppv ) = 0;
    virtual UINT32 GCCSTDC AddRef( ) = 0;
    //Destructs the SQLDriver object
    virtual UINT32 GCCSTDC Release( ) = 0;

    //Gets a new SQLConnection object
    virtual SQLResult GCCSTDC getSQLConnection (
        ppSQLConnection ppConn) = 0;

    //Set/Get options for database driver level properties

```

```

virtual SQLResult GCCSTDC setOption (
    eSQLDriverOption  eOption,
    INT32              lValue ) = 0;

virtual SQLResult GCCSTDC getOption (
    eSQLDriverOption  eOption,
    pINT32            plValue,
    INT16             iMaxLength,
    pINT16            piLength ) = 0;
};

```

SQLConnection

Implementacja klasy SQLConnection odpowiada za

- ustanowienie połączenia z bazą danych,
- pobranie obiektu SQLCommand do przetwarzania zapytań i składowanych procedur
- pobranie obiektu SQLMetaData do odczytu metadanych
- obsługę transakcji.

```

abstract COMINTF SQLConnection
{

    virtual INT32 GCCSTDC QueryInterface( GUID riid, ppVOID ppv ) = 0;
    virtual UINT32 GCCSTDC AddRef( ) = 0;
    //Destructs the SQLConnection object
    virtual UINT32 GCCSTDC Release( ) = 0;

    //Attach to a database server
    virtual SQLResult GCCSTDC connect (
        pCHAR pszServerName,
        pCHAR pszUserName,
        pCHAR pszPassword ) = 0;

    //Detach from a database server
    virtual SQLResult GCCSTDC disconnect ( ) = 0;

    //Destructs the SQLConnection object

    //Gets a new SQLCommand object
    virtual SQLResult GCCSTDC getSQLCommand (
        ppSQLCommand ppComm ) = 0;

    //Gets a new SQLMetaData object
    virtual SQLResult GCCSTDC getSQLMetaData (
        ppSQLMetaData ppMeta ) = 0;

    //Set/Get options for database connection level properties
    virtual SQLResult GCCSTDC setOption (
        eSQLConnectOption eCOption,
        INT32              lValue ) = 0;

    virtual SQLResult GCCSTDC getOption (
        eSQLConnectOption eCOption,
        pINT32            plValue,
        INT16             iMaxLength,
        pINT16            piLength ) = 0;

    //Transaction support
    virtual SQLResult GCCSTDC beginTransaction(
        UINT32 ulTrasnID );

    virtual SQLResult GCCSTDC commit(
        UINT32 ulTrasnID );

    virtual SQLResult GCCSTDC rollback(
        UINT32 ulTrasnID );

    //Error handling

```

```

virtual SQLResult GCCSTDC getErrorMessage (
    pBYTE pszError ) = 0;

virtual SQLResult GCCSTDC getErrorMessageLen (
    pUINT16 puErrorLen ) = 0;
};

```

SQLCommand

Klasa SQLCommand zapewnia metody do przetwarzania zapytań lub składowanych procedur i zwraca obiekt SQLCursor jeżeli jest on dostępny. Obsługuje również wielokrotne wykonywanie przygotowanych zapytań z wiązaniem parametrów i zwracaniem wielu cursorów ze składowanej procedury.

```

abstract COMINTF SQLCommand
{
    virtual INT32 GCCSTDC QueryInterface( GUID riid, ppVOID ppv ) = 0;
    virtual UINT32 GCCSTDC AddRef( ) = 0;
    //Destructs the SQLCommand object
    virtual UINT32 GCCSTDC Release( ) = 0;

    //Set/Get options for command level properties
    virtual SQLResult GCCSTDC setOption (
        eSQLCommandOption eSOption,
        INT32 lValue ) = 0;

    virtual SQLResult GCCSTDC getOption (
        eSQLCommandOption eSOption,
        pINT32 plValue,
        INT16 iMaxLength,
        pINT16 piLength ) = 0;

    //Set/Get methods for parameter binding support
    virtual SQLResult GCCSTDC setParameter (
        UINT16 uParameterNumber,
        STMTParamType ePType,
        UINT16 uLogType,
        UINT16 uSubType,
        INT32 lMaxPrecision,
        INT32 lMaxScale,
        UINT32 ulLength,
        pVOID pBuffer,
        BOOL bIsNull ) = 0;

    virtual SQLResult GCCSTDC getParameter (
        UINT16 uParameterNumber,
        pVOID pData,
        UINT32 ulLength,
        pINT32 plInd ) = 0;

    //Prepare and Execute SQL and Stored Procedure
    virtual SQLResult GCCSTDC prepare (
        pCHAR pszSQL ) = 0;

    virtual SQLResult GCCSTDC execute (
        ppSQLCursor ppCur) = 0;

    //Direct execution of Query or Stored Procedure or DDL
    //No parsing will be done
    virtual SQLResult GCCSTDC executeImmediate (
        pCHAR pszSQL,
        ppSQLCursor ppCur) = 0;

```

```

//A new SQLCursor object will be returned if there is another result set
//available from a stored procedure execution
virtual SQLResult GCCSTDC getNextCursor (
    ppSQLCursor ppCur);

//No.of rows affected as a result of INSERT/DELETE/UPDATE
virtual SQLResult GCCSTDC getRowsAffected (
    pINT32 plRows) = 0;

//Free resources used for parameter binding
virtual SQLResult GCCSTDC close () = 0;

//Error Handling
virtual SQLResult GCCSTDC getErrorMessage (
    pBYTE pszError ) = 0;

virtual SQLResult GCCSTDC getErrorMessageLen (
    pUINT16 puErrorLen ) = 0;
};

```

SQLCursor

SQLCursor przechowuje dane i metadane dotyczące wykonywanego zapytania albo składowanej procedury i posiada metody do pobierania wartości poszczególnych pól.

```

abstract COMINTF SQLCursor
{
    virtual INT32 GCCSTDC QueryInterface( GUID riid, ppVOID ppv ) = 0;
    virtual UINT32 GCCSTDC AddRef( ) = 0;
    //Destructs the SQLCursor object
    virtual UINT32 GCCSTDC Release( ) = 0;

    //Error handling
    virtual SQLResult GCCSTDC getErrorMessage (
        pBYTE pszError );

    virtual SQLResult GCCSTDC getErrorMessageLen (
        pUINT16 puErrorLen );

    //Metadata access methods
    virtual SQLResult GCCSTDC getColumnCount (pUINT16 puColumns) = 0;

    virtual SQLResult GCCSTDC getColumnNameLength (
        UINT16 uColumnNameNumber,
        pUINT16 puLen ) = 0;

    virtual SQLResult GCCSTDC getColumnName (
        UINT16 uColumnNameNumber,
        pCHAR pColumnName) = 0;

    virtual SQLResult GCCSTDC getColumnType (
        UINT16 uColumnNameNumber,
        pUINT16 puType,
        pUINT16 puSubType ) = 0;

    virtual SQLResult GCCSTDC getColumnLength (
        UINT16 uColumnNameNumber,
        pUINT32 pulLength ) = 0;

    virtual SQLResult GCCSTDC getColumnPrecision (
        UINT16 uColumnNameNumber,
        pINT16 piPrecision ) = 0;

    virtual SQLResult GCCSTDC getColumnScale (
        UINT16 uColumnNameNumber,
        pINT16 piScale ) = 0;

    virtual SQLResult GCCSTDC isNullable (

```

```
    UINT16 uColumnNumber,
    pBOOL pbNullable ) = 0;

virtual SQLResult GCCSTDC isAutoIncrement (
    UINT16 uColumnNumber,
    pBOOL pbAutoIncr ) = 0;

virtual SQLResult GCCSTDC isReadOnly (
    UINT16 uColumnNumber,
    pBOOL pbReadOnly ) = 0;

virtual SQLResult GCCSTDC isSearchable(
    UINT16 uColumnNumber,
    pBOOL pbSearchable ) = 0;

virtual SQLResult GCCSTDC isBlobSizeExact(
    UINT16 uColumnNumber,
    pBOOL pbBlobExactSize );

//Fetch the next record or the next set of records
virtual SQLResult GCCSTDC next() = 0;

//Data access methods
virtual SQLResult GCCSTDC getString (
    UINT16 uColumnNumber,
    pCHAR pData,
    pBOOL pbIsNull );

virtual SQLResult GCCSTDC getShort (
    UINT16 uColumnNumber,
    pINT16 pData,
    pBOOL pbIsNull );

virtual SQLResult GCCSTDC getLong (
    UINT16 uColumnNumber,
    pINT32 pData,
    pBOOL pbIsNull );

virtual SQLResult GCCSTDC getDouble (
    UINT16 uColumnNumber,
    pDFLOAT pData,
    pBOOL pbIsNull );

virtual SQLResult GCCSTDC getBcd (
    UINT16 uColumnNumber,
    pFMTBcd pData,
    pBOOL pbIsNull );

virtual SQLResult GCCSTDC getTimeStamp (
    UINT16 uColumnNumber,
    pSQLLTIMESTAMP pData,
    pBOOL pbIsNull );

virtual SQLResult GCCSTDC getTime (
    UINT16 uColumnNumber,
    pINT32 pData,
    pBOOL pbIsNull );

virtual SQLResult GCCSTDC getDate (
    UINT16 uColumnNumber,
    pINT32 pData,
    pBOOL pbIsNull );

virtual SQLResult GCCSTDC getBytes (
    UINT16 uColumnNumber,
    pBYTE pData,
    pBOOL pbIsNull );

virtual SQLResult GCCSTDC getBlobSize (
    UINT16 uColumnNumber,
    pUINT32 pullLength,
    pBOOL bIsNull );
```

```

virtual SQLResult GCCSTDC getBlob (
    UINT16 uColumnNumber,
    PVOID pData,
    PBOOL bIsNull,
    UINT32 ulLength );
};

```

SQLMetaData

Interfejs SQLMetaData definiuje rozmaite metadane bazy danych, które można pobrać z otwartego połączenia bazodanowego. Metoda getSQLMetaData() obiektu SQLConnection zwraca obiekt SQLMetaData. Obiekt SQLMetaData zwraca tylko te metadane, których potrzebuje MIDAS i komponenty dostępu do danych z Kylix, Delphi i C++Builder. Istnieje wiele innych właściwości baz danych związanych z metadanymi, które zostały pominięte dla uproszczenia implementacji. Jednak do większości z nich można się dostać przy pomocy opcji „get” i „set” obiektu SQLMetaData bez zmieniania interfejsu.

```

abstract COMINTF SQLMetaData
{
    virtual INT32 GCCSTDC QueryInterface( GUID riid, ppVOID ppv ) = 0;
    virtual UINT32 GCCSTDC AddRef( ) = 0;
    //Destructs the SQLMetaData object
    virtual UINT32 GCCSTDC Release( ) = 0;

    //Set/Get options for database metadata properties
    virtual SQLResult GCCSTDC setOption (
        eSQLMetaDataOption eMOption,
        INT32 lValue ) = 0;

    virtual SQLResult GCCSTDC getOption (
        eSQLMetaDataOption eDOption,
        pINT32 plValue,
        INT16 iMaxLength,
        pINT16 piLength ) = 0;

    //Get Object list
    virtual SQLResult GCCSTDC getObjectList (
        eSQLObjectType eObjType,
        ppSQLCursor ppCur) = 0;

    //Get Table list
    virtual SQLResult GCCSTDC getTables (
        pCHAR pszTableName,
        UINT32 uTableType,
        ppSQLCursor ppCur) = 0;

    //Get Procedure list
    virtual SQLResult GCCSTDC getProcedures (
        pCHAR pszProcName,
        UINT32 uProcType,
        ppSQLCursor ppCur) = 0;

    //Get Column metadata for a given table
    virtual SQLResult GCCSTDC getColumns (
        pCHAR pszTableName,
        pCHAR pszColumnName,
        UINT32 uColType,
        ppSQLCursor ppCur) = 0;

    //Get Parameter metadata for a given Stored Procedure
    virtual SQLResult GCCSTDC getProcedureParams (
        pCHAR pszProcName,
        pCHAR pszParamName,
        ppSQLCursor ppCur) = 0;

    //Get Index info associated with a given table
    virtual SQLResult GCCSTDC getIndices (

```

```

    pCHAR pszTableName,
    UINT32 uIndexType,
    ppSQLCursor ppCur) = 0;

//Error handling
virtual SQLResult GCCSTDC getErrorMessage (
    pBYTE pszError );

virtual SQLResult GCCSTDC getErrorMessageLen (
    pUINT16 puErrorLen );
};

```

Metody SQLMetaData

Dla wszystkich poniższych metod można wyspecyfikować wzorzec wyszukiwania dla nazwy katalogu i nazwy schematu poprzez ustawienie właściwości eMetaCatalogName i eMetaSchemaName w wywołaniu setOption().

Wzorce wyszukiwania

Wzorzec wyszukiwania może być wyspecyfikowany dla zawężenia zbioru wyników zwracanych przez poniższe metody i może zawierać następujące znaki, w oparciu o standardowe znaki zamienników SQL:

- znak podkreślenia (_) reprezentuje dowolny, pojedynczy znak
- znak procentu (%) reprezentuje sekwencję jednego lub więcej znaków
- znak ucieczki, poprzedzający znak podkreślenia lub procentu, umożliwia wykorzystywanie ich jako literały we wzorcach wyszukiwania

Sekwencja dwóch znaków ucieczki umożliwia wykorzystanie znaku ucieczki jako literału we wzorcu wyszukiwania.

Specyficzne dla bazy danych znaki ucieczki można odczytać z obiektu SQLMetaData wywołując metodę getOption(eMetaSQLEscapeChar).

getObjectList()

```

getObjectList(
    eSQLObjectType eObjType,
    ppSQLCursor ppCur)

```

Na podstawie parametru typu obiektu eSQLObjectType, metoda ta zwraca SQLCursor z listą wszystkich dostępnych obiektów w bazie danych.

W zwracanym kursorze znajdują się następujące kolumny:

kolumna	typ
1.RECNO	fldINT32
2.CATALOG_NAME	fldZSTRING
3.SCHEMA_NAME	fldZSTRING
4.OBJECT_NAME	fldZSTRING

Kolumny kursora są uporządkowane według nazwy obiektu (OBJECT_NAME).

getTables()

```
getTables(
    pCHAR pszTableNamePattern,
    UINT32 uTableType,
    ppSQLCursor ppCur)
```

Metoda ta w oparciu o parametr eSQLTableType zwraca SQLCursor z listą wszystkich tablic, widoków itp. z bazy danych. W celu otrzymania listy więcej niż jednego typu tabeli można złączyć logicznym OR jeden lub więcej parametrów eSQLTableType. W pierwszym argumencie można wyspecyfikować również wzorzec wyszukiwania nazwy tablicy. Jeżeli pierwszy parametr jest równy NULL, wówczas przy wyszukiwaniu żadne kryteria nie będą zastosowane.

W zwracanym kursorze znajdują się następujące kolumny:

kolumna	typ
1.RECNO	fldINT32
2.CATALOG_NAME	fldZSTRING
3.SCHEMA_NAME	fldZSTRING
4.TABLE_NAME	fldZSTRING
5.TABLE_TYPE	fldINT32

Kolumny kursora są uporządkowane według nazwy tablicy (TABLE_NAME).

getProcedure()

```
getProcedures (
    pCHAR pszProcNamePattern,
    UINT32 uProcType,
    ppSQLCursor ppCur)
```

Metoda ta, w oparciu o wartość parametru eSQLProcType, zwraca SQLCursor z listą wszystkich procedur (funkcji) znajdujących się w bazie danych. W pierwszym argumencie można również wyspecyfikować wzorzec wyszukiwania. W przypadku wartości NULL, do wyszukiwania nie zostaną użyte żadne kryteria.

W zwracanym kursorze znajdują się następujące kolumny:

kolumna	typ
1.RECNO	fldINT32
2.CATALOG_NAME	fldZSTRING
3.SCHEMA_NAME	fldZSTRING
4.PROC_NAME	fldZSTRING
5.PROC_TYPE	fldINT32
6.IN_PARAMS	fldINT16
7.OUT_PARAMS	fldINT16

Kolumny kursora są uporządkowane według nazwy procedury (PROC_NAME).

getColumns()

```
getColumns (
    pCHAR pszTableName,
    pCHAR pszColumnNamePattern,
    UINT32 uColType,
    ppSQLCursor ppCur)
```

Na podstawie nazwy tablicy, metoda ta zwraca SQLCursor z listą kolumn należących do tablicy. W drugim parametrze można również użyć wzorca wyszukiwania do odfiltrowania niektórych nazw kolumn. W parametrze uColType można określić więcej niż jedną wartość eSQLColType przy pomocy logicznego OR i uzyskać listę kolumn więcej niż jednego typu, jak na przykład RowId.

W zwracanym kursorze znajdują się następujące kolumny:

kolumna	typ
1.RECNO	fIdINT32
2.CATALOG_NAME	fIdZSTRING
3.SCHEMA_NAME	fIdZSTRING
4.TABLE_NAME	fIdZSTRING
5.COLUMN_NAME	fIdZSTRING
6.COLUMN_POSITION	fIdINT16
7.COLUMN_TYPE	fIdINT32
8.COLUMN_DATATYPE	fIdINT16
9.COLUMN_TYPENAME	fIdZSTRING
10.COLUMN_SUBTYPE	fIdINT16
11.COLUMN_PRECISION	fIdINT32
12.COLUMN_SCALE	fIdINT16
13.COLUMN_LENGTH	fIdINT32
14.COLUMN_NULLABLE	fIdINT16

Kolumny kursora są uporządkowane według nazwy kolumny (COLUMN_NAME).

getProcedureParams()

W oparciu o nazwę procedury metoda ta zwraca SQLCursor z listą parametrów wymaganych do wywołania procedury. W drugim parametrze można również wyspecyfikować wzorzec wyszukiwania do odfiltrowania niektórych nazw parametrów.

W zwracanym kursorze znajdują się następujące kolumny:

kolumna	typ
1.RECNO	fldINT32
2.CATALOG_NAME	fldZSTRING
3.SCHEMA_NAME	fldZSTRING
4.PROC_NAME	fldZSTRING
5.PARAM_NAME	fldZSTRING
6.PARAM_TYPE	fldINT16
7.PARAM_DATATYPE	fldINT16
8.PARAM_SUBTYPE	fldINT16
9.PARAM_TYPENAME	fldZSTRING
10.PARAM_PRECISION	fldINT32
11.PARAM_SCALE	fldINT16
12.PARAM_LENGTH	fldINT32
13.PARAM_NULLABLE	fldINT16

Kolumny kursora są uporządkowane według nazwy parametru (PARAM_NAME).

getIndices()

```
getIndices (
    pCHAR pszTableName,
    UINT32 uIndexType,
    ppSQLCursor ppCur)
```

Na podstawie nazwy tablicy metoda ta zwraca SQLCursor z listą kolumn jej indeksu. W parametrze uIndexType można przekazać więcej niż jedną wartość eSQLIndexType związane logicznym OR w celu uzyskania więcej niż jednego typu indeksu, jak „Unique” itp.

W zwracanym kursorze znajdują się następujące kolumny:

kolumna	typ
1.RECNO	fldINT32
2.CATALOG_NAME	fldZSTRING
3.SCHEMA_NAME	fldZSTRING
4.TABLE_NAME	fldZSTRING
5.INDEX_NAME	fldZSTRING
6.PKEY_NAME	fldZSTRING
7.COLUMN_NAME	fldZSTRING
8.COLUMN_POSITION	fldINT16
9.INDEX_TYPE	fldINT16
10.SORT_ORDER	fldZSTRING
11.FILTER	fldZSTRING

Kolumny kursora są uporządkowane według nazwy indeksu (INDEX_NAME).

Do interfejsu SQLMetaData można dodawać nowe metody do zwracania metadanych dotyczących przywilejów tablic i kolumn, dostępnych funkcji i wyrażeń SQL, więzów integralności referencyjnej itp.

Odzworowanie typów danych

Źródła danych reprezentują dane w różnych formatach i wspólny interfejs do nich, taki jak dbExpress, powinien zapewniać zbiór ogólnych typów danych do pobierania poszczególnych wartości pól. dbExpress definiuje zbiór logicznych typów danych, poprzez który można się dostać do większości typów danych SQL obsługiwanych przez różne bazy danych. Dla zachowania zgodności i ułatwienia dostosowania istniejących aplikacji te logiczne typy danych odpowiadają typom z BDE.

Jednak mapowanie z SQL na logiczne typy danych nie zawsze może być takie same, jak w BDE. Na przykład wprowadzono nowy typ fldDATETIME do reprezentowania stempli czasu bez utraty danych. Podobnie, dane numeryczne, które nie mieszczą się w typie rzeczywistym o podwójnej precyzji, są bezpośrednio mapowane na BCD, co eliminuje potrzebę włączania lub wyłączania BCD. Pomimo tego, że odbywa się tłumaczenie pomiędzy SQL a typami logicznymi i w drugą stronę, jednak związany z tym narzut na wydajność jest pomijalny.

Wewnętrznie dbExpress mapuje specyficzne dla bazy danych typy SQL na specyficzne dla bazy danych fizyczne typy danych i jeden lub więcej fizycznych typów danych mapuje na logiczne typy danych. W przeciwieństwie do BDE, dbExpress nie udostępnia klientom fizycznych typów danych, jednak na obecnym stadium pracy, programiści sterowników baz danych mogą również wybrać zwracanie fizycznych typów danych.

Podsumowanie

Na szczycie podstawowego interfejsu, programiści aplikacji mogą stworzyć warstwę buforującą i w ten sposób zapewnić przewijanie wynikowego zbioru danych w obu kierunkach. Połączenia z bazami danych bywają czasami bardzo kosztowne, więc można pomyśleć o zbudowaniu warstwy do utrzymywania puli połączeń. W łatwy sposób można również utworzyć interfejsy do tworzenia schematów, importu i eksportu danych między źródłami danych i wszelkich innych potrzeb związanych z dostępem do danych.

Najważniejsze jest jednak posiadanie cienkiej i prostej warstwy dostępu do danych, zapewniającej bardzo wydajną łączność z bazami danych i łatwość dostosowywania się do nowych źródeł danych.