99-43: The BetaDBC LIbrary - Reference Manual and Tutorial

Table of Contents

Copyright Notice

Mjłlner Informatics Report
MIA 99-43
August 1999

The BetaDBC Library

Introduction

This document describes the BetaDBC (Beta DataBase Connectivity) library for communicating with relational databases using SQL (Structured Query Language, [Date 93]). The library implements the pattern connection used to model connections to relational databases. This pattern contains patterns for querying and manipulating relational databases. Additional support for transactions may be added by including the fragment transactions. Scrolling cursors are implemented in the fragment scrollingresultset.

Please note that this manual and tutorial is very preliminary. The text may be wrong, and several concepts may be better explained. Sorry for any inconveniences this may impose on you.

The BetaDBC Library

BetaDBC Basics

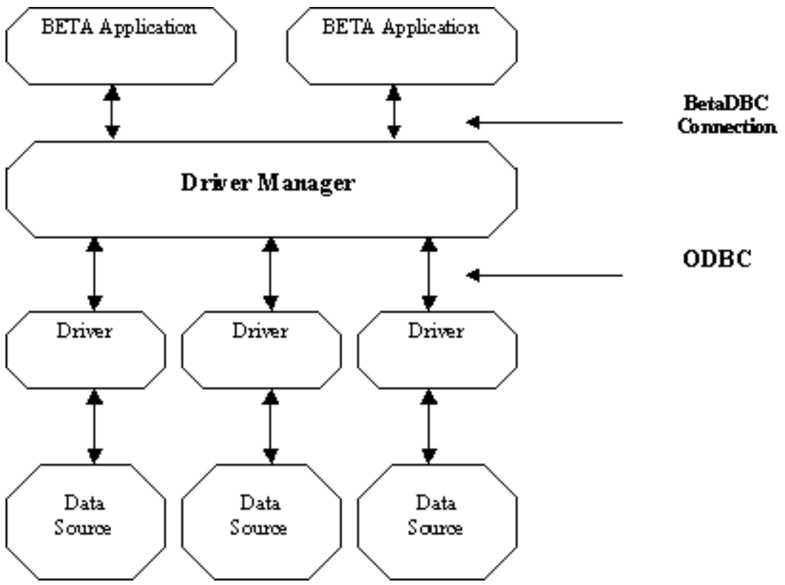
Generally speaking, there are two ways of programming to a relational database, namely using

- Embedded SQL. Using this approach SQL statements are embedded in a host language. Using a Database Management System (DBMS) specific precompiler, the application program is precompiled, transforming statements containing shared variables (variables shared by the DBMS and the application program) into DBMS library calls. After compilation the program is linked with the DBMS' runtime library.
- Call Level Interfaces (CLIs). These are libraries of functions. The functions are the native application programming interface of the DBMS or calls to it. In this way, a precompiler is not needed.

Both approaches suffer from some problems although standards have been proposed: Embedded SQL ties a program to a specific DBMS meaning that at least a new compilation will have to be made if a new DBMS is to be used. CLIs are hard to learn and often contain a lot of DBMS specific functions.

BetaDBC combines these two approaches in such a way that although shared variables may be used a precompiler is not needed and furthermore all DBMS may be treated alike. In order to achieve this BetaDBC currently builds upon and extends Open DataBase Connectivity (ODBC [Geiger 95])

This means that the architecture of a typical BetaDBC application may be outlined as below:



A BETA application typically connects to a DBMS via BetaDBC, calls BetaDBC functions, processes results and disconnects. The driver manager loads and unloads drivers as requested by applications, and processes function calls before sending them to a driver. The driver then processes the functions calls, submits SQL requests to data sources and returns results. Data sources encapsulate the data in form of tables that a user wants to access together with an associated operating system, DBMS, and network platform (if applicable) used to access the DBMS.

Details on how to create data sources and use BetaDBC in this environment will be given below.

The BetaDBC Library

The BetaDBC Interface

The BetaDBC interface is built around the concept of a connection that models a connection to a relational DBMS. The BetaDBC interface to a connection is (see below for the full interface):

```
Connection: (*)
  (#
     <<SLOT ConnectionLib:Attributes>>;
     declareVar: (*) ...;
     declareInteger: (*) declareVar ...;
     declareReal: (*) declareVar ...;
     declareText: (*) declareVar ...;
     declareBoolean: (*) declareVar ...;
     declareTime: (*) declareVar ...;
     declareDate: (*) declareTime ...;
     declareClock: (*) declareTime ...;
     formatTime:< (*) ...;
     formatDate:< (*) ...;
     formatClock:< (*) ...;
     SQLStatement: (*) ...;
     directSQLStatement: (*) SQLStatement ...;
     preparedSQLStatement: (*) directSQLStatement ...;
     resultSet: (*) ...;
     open:< (*) ...;
     close:< ...;
     connectionException:< BetaDBCException ...;
     connectionNotification:< BetaDBCNotification ...;
     private: @<<SLOT ConnectionPrivate:Descriptor>>
  #)
```

Data Sources

In order to use a connection a data source must be created (See the tutorial for specifics.). To communicate with an existing data source the user must first create an instance of the connection pattern. Calling the open method on a connection:

```
open:< (*)
  (# name: ^text;
     userName: ^text;
     password: ^text;
     openConnectionException:< BetaDBCException (#  do INNER #);
     openConnectionNotification:< BetaDBCNotification (#  do INNER #)
  enter (name[],userName[],password[])
  <<SLOT ConnectionOpen:DoPart>>
  #);
```

then makes it possible to communicate with the data source. When calling open the name of the data source must be supplied whereas user name and/or password may be omitted as appropriate. Consider as an example the statement

```
('ullman97','marius',none)->sqlCon.open
```

The statement opens a connection to the data source name "ullman97" for the user "marius" without specifying a password.

Shared Variables

Definition of shared variables are done via the "declare..." methods. To e.g. declare a text studioName as a shared variable named "studioName" use

```
studioName: @text;

...

do 'studioName'->declareText
   (# set::(# do value->studioName #);
      get::(# do studioName[]->value[] #)
   #)
```

SQL Statements

Data manipulation and definition is done using the SQL statement patterns. The SQL statement patterns have the following interfaces:

```
SQLStatement:(*)
  (# <<SLOT SQLStatementLib:Attributes>>;
     cursorType:<(*) ...;
     resultSetType:<(*) ...;
     execute:<(*) ...;
     open:(*) ...;
     close:(*) ...;
     SQLStatementException:< BetaDBCException ...;
     SQLStatementNotification:< BetaDBCNotification ...;
     get:< ...;
     set:< ...;
     private: @<<SLOT SQLStatementPrivate:Descriptor>>;
  enter set
  do INNER
  exit get
  #);

directSQLStatement:(*) SQLStatement
  (# <<SLOT DirectSQLStatementLib:Attributes>>;
     currentMarker:(*) @ ...;
     marker:(*) ...;
     b: (*) marker ...;
     c: (*) marker ...;
     d: (*) marker ...;
     f: (*) marker ...;
     i: (*) marker ...;
     s: (*) marker ...;
     t: (*) marker ...;
     setByName: (*) ...;
     setBooleanByName: (*) ...;
     setClockByName: (*) ...;
     setDateByName: (*) ...;
     setFloatByName: (*) ...;
     setIntegerByName: (*) ...;
     setTextByName: (*) ...;
     setTimeByName: (*) ...;
     execute::  ...;
     execDirectException:< BetaDBCException ...;
     execDirectNotification:< BetaDBCNotification ...;
     set::<  ...;
     getExpanded:(*) ...;
```

```
    private: @<SLOT DirectSQLStatementPrivate:Descriptor>>;
  do INNER
  #);

preparedSQLStatement:(*)
  (# ... #)
```

To use a statement stmt, one must first open it and then associate it with an SQL statement as in

```
stmt.open;
'SELECT title, length FROM Movie WHERE studioName = \'Disney\''->stmt
```

Invoking execute on stmt will then cause the SQL statement to be executed at the database. A statement should be closed after use.

A preparedSQLStatement differs from a directSQLStatement in that a prepared statement is parsed and prepared by the data source when the statement is initialised, i.e. executing

the statement above will, if stmt is a preparedSQLStatement, cause the contents of the SQL statement to be sent to the database in order for it to be parsed and prepared for future execution. If stmt is a directSQLStatement no communication with the database will occur before calling execute.

In this way a prepared SQL statement is a little slower to initialise than a direct SQL statement but much faster to execute. Use a preparedSQLStatement only when an SQL statement has to be executed several times.[1]

The contents of a directSQLStatement can be any SQL statement with embedded shared variables and/or markers, i.e. a directSQLStatement stmt may be initialised as in

```
'SELECT title,length FROM Movie WHERE studioName = :studioName AND year = %i'
  ->stmt
```

Here "studioName" is the name of a shared variable declared as shown above. The value of the %i marker may be set using the i pattern in directSQLStatement. Now suppose that the following statements have been executed

```
 'Disney'->studioName; 1990->stmt.i
```

When executing the SQL statement the SQL contents of the statement will then conceptually be

```
SELECT title,length FROM Movie WHERE studioName = 'Disney' AND year = 1990
```

i.e., before sending an SQL statement to a database the embedded shared variables and the markers are, conceptually, substituted for their current values. After execution the contents of the statement can be changed, the markers can be reset or the statement can be closed. Also, embedded shared variables and markers can be named as in

```
'SELECT title,length FROM Movie WHERE studioName = :studioName AND year = theYear%i'
  ->stmt
```

In this case, the %i marker is named "theYear" and the statement may be used as

```
'Disney'->studioName; ('theYear',1990)->stmt.setIntegerByName
```

A preparedSQLStatement is used similarly to a directSQLStatement.

Results

Executing an SQL statement stmt will yield an instance of resultSet (here rs is a reference to a resultSet):

```
rs: ^connection.resultSet

...

do stmt.execute->rs[]
```

A resultSet implements an interface to the result of an SQL query in the following way

```
resultSet:(*)
  (# <<SLOT ResultSetLib:Attributes>>;
     columnCount:(*) integerValue ...;
     rowCount:(*) integerValue ...;
     column: (*)...;
     getColumn: (*)...;
     getColumnByName:(*) ...;
     cursorName:(*) ...;
     tuple:(*)
       (# <<SLOT ResultLib:Attributes>>;
          marker:(*) ...;
          b: marker ...;
          c: marker ...;
          d: marker ...;
          f: marker ...;
          i: marker ...;
          s: marker ...;
          t: marker ...;
          private: @...
       #);
     EOT: (*) ...;
     set:(*)
       (# varNotDeclared:<(*) exception ...;
          columnNotFound:<(*) exception ...;
          pattern: ^text
       #);
     fetch:(*) ...;
     scan:(*)
       (# current: ^tuple;
          varNotDeclared:<(*) exception ...;
          columnNotFound:<(*) exception ...;
          pattern: ^text
       enter pattern[]
       ...
       #);
     resultSetException:< BetaDBCException ...;
     resultSetNotification:< BetaDBCNotification ...;
     private: @...
  #)
```

Given a resultSet, the scan method iterates over the tuples in the resultSet. There are three distinct ways to control the scan. First, one may simply execute

```
rs.scan(# ... #)
```

In the do-part of the scan one may then refer to the values of the columns in the result. This is done sequentially by referring to the markers of the current result. If, e.g., rs was retrieved as shown above,

```
rs.scan(# do current.s -> putline; current.i->putint; newline #)
```

will scan over the results in the resultSet and print the values of their columns on the screen.

Second, one may enter a string when evaluating a scan pattern as in:

```
':title %i'->rs.scan(# do title->putline; current.i->putint; newline #)
```

Here title is a shared variable named "title". This statement prints the same as above but by providing an input string it is here specified that the first column of each result should be assigned to the shared variable "title" and that the second column of each result is an integer that will be fetched via the i marker. In general one may in this way specify how each column of a result should be treated.

The two ways of scanning shown above may, in some circumstances, be problematic in that they assume a specific ordering of columns in the results. Therefore, the third way of doing a scan names the columns in the resultSet, as in e.g.:

```
'length%i title:title'->rs.scan
   (# do title->putline; current.i->putint; newline #)
```

In this way the order of the columns in the result may be changed from "title, length" to "length, title" without any problems for the last way of scanning.
Results can also be fetched one at a time by using "fetch" instead of "scan". This is, e.g., useful for interleaving the processing of two result sets. For each result set there is a pointer pointing to a current result. If rs is is a resultSet and current is a result,

```
rs.fetch -> current[]
```
will fetch the current result and advance the pointer. "current" may be used as shown above.

---

[1] Note that, currently, a preparedSQLStatement is implemented as a directSQLStatement.

Transactions

Many data sources support the use of transactions. In order to use this capability from BetaDBC. the fragment transactions must be included. An outline of the interface of this fragment is shown below. The full interface may be seen in section 5.3.

```
ORIGIN 'betadbc';
BODY 'private/transactionsbody';
-- connectionLib: Attributes --
transactionsSupported:(*)
  (# isSupported: @boolean
  ...
  exit isSupported
  #);
autoCommitMode:(*)
  (# autoCommit: @boolean
  enter (#  enter autoCommit ... #)
  exit
    (# ...
    exit autoCommit
    #)
  #);
readUncommitted: (*) integerValue (# ... #);
readCommitted: (*) integerValue (# ... #);
repeatableRead: (*) integerValue (# ... #);
serializable:(*) integerValue (# ... #);
transactionLevelSupported:(*) booleanValue
  (# level: (*) @integer
  enter level
  ...
  #);
transactionLevel:(*)
  (# level: (*) @integer
  enter (#  enter level ... #)
  exit
    (# ...
    exit level
    #)
  #);
commit:(*) (#  ... #);
rollBack:(*) (#  ... #)
```

The scope of a transaction is a whole connection including all statements allocated in it. The default is that every execution of an SQL statement starts a new transaction and automatically commits the effects of this statement after the statement has completed. This auto commit mode may be changed to manual commit mode by evaluating

```
false->sqlCon.autoCommitMode
```

where sqlCon is an instance of a connection. Note that this is only meaningful if transactionsSupported evaluates to true. In manual commit mode a series of database manipulations may be committed by executing commit. Equivalently a series of database manipulations may be aborted by executing rollBack.

Four transaction isolation levels (as defined in the SQL standard) are available, namely readUncommitted, readCommitted, repeatableRead, and serializable. Whether a transaction isolation level, such as serializable, is supported in a given connection may be checked by evaluating e.g.

```
serializable->sqlCon.transactionLevelSupported
```

Scrolling Cursors

Commonly, a result set is scanned from the beginning to the end. Sometimes, however, other orders of fetching of results are desirable. Scrolling result sets support this.

To use scrolling cursors and result sets, include the fragment scrollingresultset. Applications can investigate whether a connection supports scrolling cursors by calling scrollingCursorsSupported on a connection.

Suppose the connection con supports scrolling cursors. Then the declaration of a statement stmt as

```
stmt: @con.directSQLStatement
        (#
           cursorType::
             (# scrollable::  (#  do true->value #) #);
           resultSetType:: con.scrollingResultSet
         #)
```

will create a scrolling result set whenever stmt is executed:

```
rs: ^connection.scrollingResultSet;

...

do ...; stmt.execute->rs[]; ...
```

The result set, rs, may now be used to scroll through the result set using the methods shown below.

```
scrollingResultSet: (*) resultSet
  (#
     fetchFirst: (*) ...;
     fetchLast: (*) ...;
     fetchNext: (*) ...;
     fetchPrior: (*) ...;
     fetchRelative: (*) ...;
     fetchAbsolute: (*) ...;
     scanReverse: (*) ...;
     ...
  #)
```

fetchLast, e.g., will fetch the last tuple in the result set and scanReverse will scan through the result set backwards.

Please note that it is not possible to use both fetch and the methods declared on scrollingResultSet on the same result set.

Tutorial

The sample programs shown in the tutorial may be found in the tutorial directory accompanying BetaDBC. The examples use a database schema and examples from [Ullman 97]. It supposes that the reader is familiar with basic SQL and focuses on teaching the essentials of using BetaDBC. All examples take (up to) three command line arguments: a data source, a user name, and a password. If the data source used permits, the user name and/or password may be omitted.

Creating a Data Source

In order to use BetaDBC a suitable data source must be created. Currently, data sources are created outside BetaDBC. Different procedures must be followed depending on the operating system used. This will be changed in a future release of BetaDBC.

Creating a Data Source on Windows 95 and NT

In the ’Start’ menu choose ’Settings’ and ’Control Panel’. Start the ODBC (32 Bit) application from the ’Control Panel’. Press ’Add...’ and choose the database driver that you want to use, then press ’Finish’. Follow the driver specific instructions.

Creating a Data Source on Unix

You will need a .odbc.ini file in your home directory. Create such a file if it does not exist.

Suppose a driver for the PostgreSQL database is located in "/usr/local/lib/libcliPG.so". If you want to connect to the database "marius" on the database server "delirium" insert the following in your .odbc.ini file (’ReadOnly = 0’ tells the driver that you want to manipulate the "marius" database)

```
[ullman97]
# data source containing the ’marius’ db on delirium
# this database contains the movie tables from ullman97
Driver = /usr/local/lib/libcliPG.so
Database = marius
Servername = delirium
ReadOnly = 0
```

This defines a data source named "ullman97", that you may use BetaDBC to connect to. Each data source at least specifies which ODBC. In the example "ReadOnly" is a driver specific attribute of the data source "ullman97". See the documentation for the drivers used for definitions of driver specific attributes.

In the following it is assumed that a suitable data source named "ullman97" has been created.

Creating a Database

Check this section with the new setup

The next step will then be to create and insert values into a database.
Let's use the following sample database schema

```
Movie (title, year, length, inColor, studioName, producerCNo)
StarsIn (movieTitle, movieYear, starName)
MovieStar (name, address, gender, birthdate)
MovieExec (name, address, certNo, netWorth)
Studio (name, address, presCNo)
```

A series of SQL statements creating the database schema may be found in createmoviedbtables.txt. The text files used in this section are shown in section 6.

Create the tables corresponding to this schema by running the executesqlfile program also found in the tutorials directory:

```
[postgres@delirium tutorial]$ ./executesqlfile createmovie.txt ullman97 marius
```

How the executesqlfile program is implemented will be discussed later.
You may now insert some values in the database by running

```
[postgres@delirium tutorial]$ ./executesqlfile moviedbtables.txt ullman97 marius
```

The tables may later be deleted by running

```
[postgres@delirium tutorial]$ ./executesqlfile dropmovies.txt ullman97 marius
```

Querying and Retrieving from the Database

This section will introduce the basics of BetaDBC: connecting to data sources, executing simple queries and retrieving the results.

Consider the simple SQL statement

```
SELECT *
FROM Movie
WHERE studioName = 'Disney' AND year = 1990;
```

An application that uses BetaDBC, executes the above query and retrieves the result may look like:

```
ORIGIN '../betadbc';
-- program: Descriptor --
(#
   sqlCon: @connection;
   stmt: @sqlCon.directSqlStatement;
   rs: ^sqlCon.resultSet
do
   (2->arguments,3->arguments,4->arguments)->sqlCon.open;
   'SELECT title, length FROM Movie WHERE studioName  = \'Disney\' AND year = 1990'
     ->stmt.open;
```

```
   stmt.execute->rs[];
   rs.scan
     (#
     do
        'title: '->puttext;
        current.s->putline;
        'length: '->puttext;
        current.i->putint;
        newline
     #);
   stmt.close
   sqlCon.close
#)
```

The program starts out by declaring a connection, a directSQLStatement belonging to that connection and a resultSet belonging to that connection. The connection is used in order to connect to a data source in the first line of the program:

```
(2->arguments,3->arguments,4->arguments)->sqlCon.open;
```

Connection's open method takes as arguments a name of the connection, a username and a password. Thus an invocation of the program like

```
[postgres@delirium tutorial]$ ./simple ullman97 marius foobar
```

means that the first statement will be an attempt to connect the user "marius" with password "foobar" to the data source named "ullman97". If this succeeds

```
'SELECT title, length FROM Movie WHERE studioName = ''Disney'' AND year = 1990'
     ->stmt.open;
```

will open the directSQLStatement "stmt" and set its content to the query we want to execute. Executing the query yields a resultSet holding a cursor for the result

```
stmt.execute->rs[];
```

The resultSet may then be scanned. During the scan 'current' will hold a reference to a tuple in the resultSet. The values of this result may then be accessed consecutively by using the marker attributes

```
 rs.scan
     (#
     do
        'title: '->puttext;
        current.s->putline;
        'length: '->puttext;
        current.i->putint;
        newline
     #);
```

Finally, in order to free resources, the directSQLStatement and the connection are closed.

Executesqlfile

The simple scheme presented in the last section can now be used for implementing the executesqlfile program. The executeLoop shows how to reuse an SQLStatement by simply replacing it's textual contents

```
executeTxt[]->sqlCon.stmt
```

Embedded SQL - Using Shared Variables

Using shared variables makes it possible to use the values of BETA objects in place of a concrete value in SQL statements. Since no preprocessor is used by BetaDBC, it is necessary to declare shared variables imperatively, as in

```
   sqlCon:@connection
   studioName:@text;
do ...;
   'studioName'
     ->sqlCon.declareText
       (# set:: (#  do value->studioName #);
          get:: (#  do studioName[]->value[] #)
       #);
   ...
```

Here a shared text variable named "studioName" is declared. The set pattern is final bound to describe how the shared variable's value is to be set. get is final bound to describe how the value of the shared variable is to be fetched.

Then, using embedded SQL syntax, one may use shared variables in SQL statements:

```
   stmt:@sqlCon.directSQLStatement;
do ...;
   'INSERT INTO Studio(name, address) VALUES (:studioName, :studioAddr)'
     ->stmt.open;
   ...
```

This means that when executing stmt, ":studioName" and ":studioAddr" will (conceptually) be replaced by the values of the BETA text variables "studioName" and "studioAddr", and the resulting SQL statement will then be executed.

Using BetaDBC it is possible to declare most commonly used objects as shared variables (i.e., boolean, integer, real, text, date and time).
The figure below shows a full program that will execute the statement above. "stmt.getExpanded" returns in a text how the SQL statement would look if it was executed at that point.

```
ORIGIN '../betadbc';
-- program: Descriptor --
(# sqlCon: @connection;
   stmt: @sqlCon.directSqlStatement;
   studioName,studioAddr: @text
do (2->arguments,3->arguments,4->arguments)->sqlCon.open;
   'studioName'
     ->sqlCon.declareText
```

```
        (# set:: (#  do value->studioName #);
           get:: (#  do studioName[]->value[] #)
        #);
    'studioaddr'
      ->sqlCon.declareText
        (# set:: (#  do value->studioAddr #);
           get:: (#  do studioAddr[]->value[] #)
        #);
    'Input a studio name: '->puttext;
    getLine->studioName.puttext;
    ' and address: '->puttext;
    getLine->studioAddr.puttext;
    'INSERT INTO Studio(name, address) VALUES (:studioName, :studioAddr)'
      ->stmt.open;
    stmt.getExpanded->putline;
    stmt.close;
    sqlCon.close
#)
```

Embedded SQL - Fetching Results

Suppose that we are executing a statement that return a result. Embedded SQL can then also be used to fetch results directly into shared variables. In BetaDBC this is done through the use of the scan pattern. Suppose we are executing

```
'SELECT MovieExec.name, netWorth FROM Studio, MovieExec
 WHERE presCNo = certNo AND Studio.name = :studioName'
```

Then,

```
':presName :presNetWorth'
     ->(stmt.execute).scan(# ... #)
```

will cause the first column in each result tuple to be assigned to the shared integer variable "presName", and the second column to "presNetWorth". The full code is shown below:

```
ORIGIN '../betadbc';
INCLUDE '~beta/basiclib/formatio';
-- program: Descriptor --
(# sqlCon: @connection;
   stmt: @sqlCon.directSqlStatement;
   studioName,presName: @text;
   presNetWorth: @integer
do 'studioName'
     ->sqlCon.declareText
       (# set::  (#  do value->studioName #);
          get::  (#  do studioName[]->value[] #)
       #);
   'presName'
     ->sqlCon.declareText
       (# set::  (#  do value->presName #);
          get::  (#  do presName[]->value[] #)
       #);
   'presNetWorth'
     ->sqlCon.declareInteger
       (# set::  (#  do value->presNetWorth #);
          get::  (#  do presNetWorth->value #)
       #);
   (2->arguments,3->arguments,4->arguments)->sqlCon.open;
```

```
   'Input a studio name: '->puttext;
   getLine->studioName.puttext;
   'SELECT MovieExec.name, netWorth FROM Studio, MovieExec WHERE presCNo = certNo AND Studio.na
     ->stmt.open;
   stmt.getExpanded->putline;
   ':presName :presNetWorth'
     ->(stmt.execute).scan
       (#
       do 'The net worth of the president %s \nof %s is %i $\n'
            ->putFormat
              (#  do presName[]->s; studioName[]->s; presNetWorth->i #)
       #);
   stmt.close;
   sqlCon.close
#)
```

An Ad-hoc Query Evaluator

We now have most of the building blocks to create an ad-hoc query evaluator, i.e., a program that connects to a data source and in a loop prompts for SQL statements that are to be executed on this data source. The following implements such a program.

As long as the user inputs anything but an empty line this input is sent to the data source as an SQL statement:

```
getline->stmt;
(if (stmt).empty then leave L if);
stmt.execute->res[];
```

If successful, the result is examined. First the column information of the resultSet is extracted:

```
(for j: res.columnCount repeat
  '%s: %s\t'->putFormat
              (#
              do (j->res.getColumn).name[]->s;
                 (j->res.getColumn).dataTypeName[]->s
              #)
for);
```

Each resultSet has a columnCount yielding the number of columns in the resultSet. For each column, information such as name and dataTypeName (a DBMS specific datatype name) may be retrieved.

If the columnCount is non-zero, the results are fetched:

```
(if res.columnCount > 0 then
    res.scan
      (#
      do (for i: res.columnCount repeat
              (if (i->res.getColumn).DataType##
               // text## then
                  current.s->puttext
               // integerObject## then
                  current.i->putint
               // realObject## then
                  current.f->putreal
               // booleanObject## then
                  (if current.b then
                      'true'->puttext;
```

```
                    else
                       'false'->puttext
                   if)
                // time## then
                   current.t->puttime
                else
                   'Unknown data type!!!'->puttext
               if);
               '\t'->puttext
          for);
          newline
       #)
    else
       'DML/DDL statement executed successfully!'->putLine
if)
```

Again the information about columns in the resultSet is used. By evaluating

```
(i->res.getColumn).DataType##
```

the BETA pattern corresponding to the SQL datatype in column i is found.

The BetaDBC Library

Text Files for the Tutorial

createmoviedbtables.txt

```
CREATE TABLE MovieStar (
    name CHAR(30),
    address VARCHAR(255),
    gender CHAR(1),
    birthdate INTEGER
)

CREATE TABLE Movie (
    title VARCHAR(255),
    year INTEGER,
    length INTEGER,
    inColor CHAR(1),
    studioName CHAR(50),
    producerCNo INTEGER
)

CREATE TABLE StarsIn (
    movieTitle VARCHAR(255),
    movieYear INTEGER,
    starName CHAR(30)
)

CREATE TABLE MovieExec (
    name CHAR(30),
    address VARCHAR(255),
    certNo INTEGER,
    netWorth INTEGER
)

CREATE TABLE Studio (
    name CHAR(50),
    address VARCHAR(255),
    presCNo INTEGER
)
```

deletemoviedbtables.txt

```
DROP TABLE MovieStar

DROP TABLE Movie

DROP TABLE StarsIn

DROP TABLE MovieExec

DROP TABLE Studio
```

populatemoviedbtables.txt

```
INSERT INTO Movie VALUES (’Star Wars’, 1990, 120, ’y’, ’Disney’, 1)

INSERT INTO Movie VALUES (’Pretty Woman’, 1990, 119, ’y’, ’Disney’,999)

INSERT INTO Studio VALUES (’Disney’, ’Drive Rd.’, 1)
```

```
INSERT INTO Studio VALUES ('Paramount', 'Drive Way', 2)

INSERT INTO MovieExec VALUES ('Marius', 'Home', 1, 3000000)

INSERT INTO MovieExec VALUES ('Lucas', 'Home', 2, 2000000)
```

References

[Date 93] Date, C.J. and Darwen, H., A Guide to the SQL Standard, Addison Wesley, Reading, MA, 1993.

[Geiger 95] Geiger, K. Inside ODBC, Microsoft Press, 1995.

[Ullman 97] Ullman, J.D., Widom, J., A First Course in Database Systems, Prentice Hall International, 1997.

Betadbc Interface

```
ORIGIN '~beta/basiclib/betaenv';
INCLUDE '~beta/basiclib/timedate';
BODY 'private/betadbcbody';
-- lib: Attributes --
Connection: (* A connection to a relational DBMS *)
  (#
     <<SLOT ConnectionLib:Attributes>>;
     declareVar:
     (* Used to declare a wrapper of a BETA object so that
      * the object may be used in SQLStatements *)
       (#
          <<SLOT DeclareVarLib:Attributes>>;
          varName: ^text;
          set:<
          (* Furtherbind this to set the value of the object that you wrap *)
           object;
          get:<
          (* Furtherbind this to get the value of
           * the object that you wrap *) object;
          setAsText:<
          (* Furtherbind in concrete subpatterns to set
           * the wrapped object's value to the value 'value' *)
            (# value: ^text
            enter value[]
            ...
            #);
          getAsText:<
          (* Furtherbind in concrete subpatterns to get
           * the wrapped object's value as a text *)
            (# value: ^text
            ...
            exit value[]
            #)
       enter varName[]
       ...
       exit varName[]
       #);
     declareInteger: (* Declares a wrapper around an integer *) declareVar
       (#
          <<SLOT DeclareIntegerLib:Attributes>>;
          set::<
            (# value: @integer
            enter value
            ...
            #);
          get::<  (# value: @integer do INNER exit value #);
          setAsText::<  (#  ... #);
          getAsText::<
            (#
            ...
            #)
       #);
     declareReal: (* Declares a wrapper around a real *) declareVar
       (#
          <<SLOT DeclareRealLib:Attributes>>;
          set::<
            (# value: @real
            enter value
            do INNER
            #);
          get::<
            (# value: @real
```

```
        do INNER
        exit value
        #);
      setAsText::<
        (#
        ...
        #);
      getAsText::<  (#  ... #);
   #);
 declareText:
 (* Declares a wrapper around a text *) declareVar
   (#
      <<SLOT DeclareTextLib:Attributes>>;
      set::<
        (# value: ^text
        enter value[]
        do INNER
        #);
      get::<
        (# value: ^text
        do INNER
        exit value[]
        #);
      setAsText::<
        (#
        ...
        #);
      getAsText::<  (#  ... #);
   #);
 declareBoolean:
 (* Declares a wrapper around a boolean *) declareVar
   (#
      <<SLOT DeclareBooleanLib:Attributes>>;
      set::<
        (# value: @boolean
        enter value
        do INNER
        #);
      get::<
        (# value: @boolean
        do INNER
        exit value
        #);
      setAsText::<
        (#
        ...
        #);
      getAsText::<  (#  ... #);
   #);
 declareTime: (* Declares a wrapper around a ’time’ *) declareVar
   (#
      <<SLOT DeclareTimeLib:Attributes>>;
      set::<
        (# value: @time
        enter value
        do INNER
        #);
      get::<
        (# value: @time
        do INNER
        exit value
        #);
      setAsText::<
        (#
        ...
        #);
```

```
    getAsText::<  (#  ... #);
  #);
declareDate:
(* Declares a wrapper around a 'date'. Only the year, month and day
 * attributes of the time value object are taken into account
 *) declareTime (# <<SLOT DeclareDateLib:Attributes>> #);
declareClock:
(* Declares a wrapper around a 'time'. Only the hour, minute and sec
 * attributes of the time value object are taken into account *)
 declareTime (# <<SLOT DeclareClockLib:Attributes>> #);
formatTime:<
(* Called by BetaDBC to format 'time' values
 * Furtherbind this if a date format other than
 * 'YYYY-MM-DD HH:MM:SS' is needed *)
  (# t: @time; value: ^text
  enter t
  ...
  exit value[]
  #);
formatDate:<
(* Called by BetaDBC to format 'date' values
 * Furtherbind this if a date format other than
 * 'YYYY-MM-DD' is needed *)
  (# d: @time; value: ^text
  enter d
  ...
  exit value[]
  #);
formatClock:<
(* Called by BetaDBC to format 'clock' values
 * Furtherbind this if a clock format other than
 * 'HH:MM:SS' is needed *)
  (# c: @time; value: ^text
  enter c
  ...
  exit value[]
  #);
SQLStatement:
(* The abstract superpattern for all SQL statements.
 * To use an instance of a subclass of SQLStatement:
 *    1. Open the statement by calling 'open',
 *      1a. specifying an enter parameter to, or
 *      1b. setting the statement's SQL contents by evaluating the enter part.
 *    2. Call 'execute'.
 *    3. Go to 1b., if necessary
 *    4. Call 'close'
 *)
  (#
    <<SLOT SQLStatementLib:Attributes>>;
    cursorType:<
    (* The type of cursor that is created when
     * an SQLStatement is executed *)
      (#
        insensitive:<
        (* Should be furtherbound to set value to 'true'
         * if the cursor should be insensitive to concurrent changes *)
         booleanValue;
        readOnly:<
        (* Furtherbind to to set value to 'false' if modification by cursor is needed *)
        booleanValue (#  do true->value; INNER #);
        scrollable:<
        (*
         Furtherbind this if the resulting resultSet should be scrollable

         *) booleanValue;
        name:<
```

```
            (* Furtherbind this to set the name of the cursor.
             If setName is not furtherbound a default name will be generated. *)
            (# value: ^text do INNER exit value[] #)
         #);
       resultSetType:<
       (* The type of resultSet that will be opened
        * when executing this(SQLStatement) *) resultSet;
       execute:< (* Executes this(SQLStatement) *)
         (# res: ^resultSetType
         ...
         exit res[]
         #);
       open:
       (* Opens this(SQLStatement).
        * An SQLStatement must be opened before use *)
         (# value: ^text
         enter value[]
         ...
         #);
       close:
       (* Closes this(SQLStatement).
        * Call close when done with this(SQLStatement) *)
         (#  ... #);
       SQLStatementException:<
        BetaDBCException (# do INNER #);
       SQLStatementNotification:< BetaDBCNotification (# do INNER #);
       get:< (# t: ^text ... exit t[] #);
       set:<
         (# t: ^text
         enter t[]
         ...
         #);
       private: @...
    enter set
    do INNER
    exit get
    #);
  directSQLStatement:
  (* Use this statement type if a statement will be executed at most
   * a few times. The contents may contain variable placeholders in the form
   *    :varname
   * designating that the shared variable named 'varname'
   * will be bound to that place. In addition, also
   *    %b for booleans
   *    %c for clocks
   *    %d for dates
   *    %f for reals
   *    %i for integers
   *    %s for texts
   *    %t for time
   * may be used when setting the contents of this(directSQLStatement)
   * Furthermore, placeholders may be named as in
   *    aname%f or aname:varname
   * signifying that the value of the %f or :varname placeholder may be set/get
   * by calling setByName/getByName
   * with "aname" as parameter
   *) SQLStatement
    (#
       <<SLOT DirectSQLStatementLib:Attributes>>;
       currentMarker:
       (* The current marker decides which non-variable placeholder will be set
        * if one of the subpatterns of "marker" is used
        *)
         (#
            set:
              (#
```

```
              enter no
              ...
              #);
            get:
              (#
              ...
              exit no
              #);
            no: @integer
         enter set
         exit get
         #);
       marker:
       (* Abstract superpattern for markers.
        * Advances currentMarker by 1
        *) (# t: ^text ... #);
       b:
       (* Set placeholder number 'currentMarker'
        * to the boolean 'value' *) marker
         (# value: @boolean
         enter value
         ...
         #);
       c:
       (* Set placeholder number 'currentMarker'
        * to the clock 'value' *) marker
         (# value: @time
         enter value
         ...
         #);
       d:
       (* Set placeholder number 'currentMarker'
        * to the date 'value' *) marker
         (# value: @time
         enter value
         ...
         #);
       f:
       (* Set placeholder number 'currentMarker'
        * to the real 'value' *) marker
         (# value: @real
         enter value
         ...
         #);
       i:
       (* Set placeholder number 'currentMarker'
        * to the integer 'value' *) marker
         (# value: @integer
         enter value
         ...
         #);
       s:
       (* Set placeholder number 'currentMarker'
        * to the text 'value' *) marker
         (# value: ^text
         enter value[]
         ...
         #);
       t:
       (* Set placeholder number 'currentMarker'
        * to the time 'value' *) marker
         (# value: @time
         enter value
         ...
         #);
       setByName:
```

```
(* Abstract superpattern for patterns that
 * sets the value of a named placeholder *)
  (#
     name: ^text;
     t: ^text;
     nameNotFound:< exception
       (#
       ...
       #)
  enter name[]
  ...
  #);
setBooleanByName:
(* Set placeholder named 'name'
 * to the boolean 'value' *) setByName
  (# value: @boolean
  enter value
  ...
  #);
setClockByName:
(* Set placeholder named 'name'
 * to the clock 'value' *) setByName
  (# value: @time
  enter value
  ...
  #);
setDateByName:
(* Set placeholder named 'name'            * to the date 'value' *)
 setByName
  (# value: @time
  enter value
  ...
  #);
setFloatByName:
(* Set placeholder named 'name'
 * to the real 'value' *) setByName
  (# value: @real
  enter value
  ...
  #);
setIntegerByName:
(* Set placeholder named 'name'
 * to the integer 'value' *) setByName
  (# value: @integer
  enter value
  ...
  #);
setTextByName:
(* Set placeholder named 'name'
 * to the text 'value' *) setByName
  (# value: ^text
  enter value[]
  ...
  #);
setTimeByName:
(* Set placeholder named 'name'
 * to the time 'value' *) setByName
  (# value: @time
  enter value
  ...
  #);
execute::
  (#
  ...
  #);
execDirectException:< BetaDBCException (# do INNER #);
```

```
    execDirectNotification:< BetaDBCNotification (# do INNER #);
    private: @...;
    set::<
      (#
         varNotDeclared:<
          exception
           (# name: ^text
           enter name[]
           ...
           #)
      ...
      #);
    getExpanded:
    (* Get the contents of this(directSQLStatement)
     * as it would appear if the statement was executed now *)
      (# value: ^text
      ...
      exit value[]
      #)
 do INNER
 #);
preparedSQLStatement:
(* Use this statement type if a statement will be executed multiple
 * times with different bindings.
 * ONLY IMPLEMENTED AS A DIRECT STATEMENT
 *) directSQLStatement
  (# <<SLOT preparedSQLStatementLib:Attributes>> do INNER #);
resultSet:
(* A result of an SQLStatement.
 * If columnCount <> 0 then the resultSet is can be scanned or fetched from.
 * The tuples of a resultSet can be read at most once.
 *)
  (#
    <<SLOT resultSetLib:Attributes>>;
    columnCount:
    (* The number of columns in this(resultSet) *) integerValue
      (#  ... #);
    tupleCount:
    (* If the statement that created this(resultSet) was a
     * 1. INSERT, DELETE or UPDATE statement
     *     tupleCount yield the number of tuples
     *     affected by the statement
     * 2. SELECT statement
     *     tupleCount MAY (i.e. does not in all circumstances) yield
     *     the number of rows selected
     *) integerValue (#  ... #);
    column:
    (* A column in this(resultSet) *)
      (#
         name: ^Text;
         no: @Integer;
         dataType:
         (* The BETA pattern corresponding to the SQL datatype
          * for this column. If the SQL datatype is DATE, TIME or
          * TIMESTAMP then the corresponding BETA pattern will
          * be time
          *) ##Object;
         dataTypeName: (* DBMS specific type name *) ^Text;
         dataTypeNo: (* ODBC specific numbering of SQL datatypes *)
           @integer;
         nullAble: @Boolean
      #);
    getColumn: (* Gets the column number 'i' in this(resultSet) *)
      (# i: @Integer; res: ^column
      enter i
      ...
```

```
    exit res[]
    #);
  getColumnByName:
  (* Gets the column designated by 'name' in this(resultSet) *)
    (#
       name: ^text;
       res: ^column;
       nameNotFound:< exception
         (#  ... #)
    enter name[]
    ...
    exit res[]
    #);
  cursorName:
  (* Gets the name of the cursor that points to this(resultSet) *)
    (# value: ^text
    ...
    exit value[]
    #);
  tuple:
  (* A row in this(resultSet). If this(resultSet) has been set
   * with a value that contained non-variable placeholders,
   * the values may be retrieved by using the markers below
   *)
    (#
       <<SLOT ResultLib:Attributes>>;
       marker:
       (* Gets the value of a non-variable placeholder and advances
        * the placeholder currently referred to
        *) (#  ... #);
       b: marker
         (# value: @boolean
         ...
         exit value
         #);
       c: marker
         (# value: @time
         ...
         exit value
         #);
       d: marker
         (# value: @time
         ...
         exit value
         #);
       f: marker
         (# value: @real
         ...
         exit value
         #);
       i: marker
         (# value: @integer
         ...
         exit value
         #);
       s: marker
         (# value: ^text
         ...
         exit value[]
         #);
       t: marker
         (# value: @time
         ...
         exit value
         #);
       private: @...
```

```
  #);
EOT:
(* If current = EOT then no more tuples are
 * available in this(resultSet) *)
  (# value: ^tuple ... exit value[] #);
set:
(* May called before using fetch on this(resultSet)
 * The pattern entered may either
 * contain
 * 1. Named columns of the form
 *          'name1:var1 name2:%i name3:var2'
 *     which means that the column named 'name1'('name3')
 *     is bound to the variable named 'var1'('var2') and the
 *     column named 'name2' may be retrieved from the
 *     current tuple using the 'i'-marker,
 * or
 * 2. Consecutive columns of the form
 *          ':var1 %i :var2'
 *     which means that the first (third) column in each result
 *     is bound to the variable named 'var1'('var3') and that
 *     the value of the second column may be retrieved from the
 *     current tuple using the 'i'-marker
 *)
  (#
     varNotDeclared:<
     (* Raised if variable used in the entered
      * pattern was not found *) exception
       (# name: ^text
       enter name[]
       ...
       #);
     columnNotFound:<
     (* Raised if this is a named scan and
      * named column was not found *) exception
       (# name: ^text
       enter name[]
       ...
       #);
     pattern: ^text
  enter pattern[]
  ...
  #);
fetch:
(* Get the current tuple of the relation over which the resultSet ranges.
 * If no more tuples exist, 'current' will be EOT
 * Advance current tuple if possible *)
  (# result: ^tuple
  ...
  exit result[]
  #);
scan:
(* Scans over the tuples of this(resultSet) starting from the current tuple.
 * A text pattern may be entered. See 'set' for details.
 *)
  (#
     current: ^tuple;
     varNotDeclared:<
     (* Raised if variable used in the entered
      * pattern was not found *) exception
       (# name: ^text
       enter name[]
       ...
       #);
     columnNotFound:<
     (* Raised if this is a named scan and
      * named column was not found *) exception
```

```
                 (# name: ^text
                 enter name[]
                 ...
                 #);
               pattern: ^text
            enter pattern[]
            ...
            #);
          resultSetException:< BetaDBCException (# do INNER #);
          resultSetNotification:< BetaDBCNotification (# do INNER #);
          private: @...
       #);
     open:<
     (* Opens this(connection). The name of the connection
      * to be opened must be supplied.
      * Supplying userName and/or password is voluntary.
      *)
       (#
          name: ^text;
          userName: ^text;
          password: ^text;
          openConnectionException:< BetaDBCException (# do INNER #);
          openConnectionNotification:< BetaDBCNotification (# do INNER #)
       enter (name[],userName[],password[])
       ...
       #);
     close:<
       (#
          closeException:< BetaDBCException
            (# do INNER #);
          closeNotification:< BetaDBCNotification
            (# do INNER #)
       ...
       #);
     connectionException:< BetaDBCException (# do INNER #);
     connectionNotification:< BetaDBCNotification (# do INNER #);
     private: @...
  #);
BetaDBCException:
(* Low level interface for catching exceptions.
 * A general exception message is supplied in msg,
 * SQL states and native error codes in SQLState,
 * NativeError in a comma-separated list.
 *) Exception
  (#
     SQLState: @text;
     NativeError: @text;
     HandleType: @integer;
     Handle: @integer
  enter (HandleType,Handle)
  ...
  #);
BetaDBCNotification: BetaDBCException
  (#  do true->continue; INNER #)
```

Scrollingresultset Interface

```
ORIGIN 'betadbc';
BODY 'private/scrollingresultsetbody';
-- connectionLib: Attributes --
scrollingResultSet:
(* A scrolling resultSet supports relative and absolute
 * positioning in the tuples of a resultSet.
 * Fetch and scan *cannot* be used on a 'scrollingResultSet' if
 * the methods defined in scrollingResultSet has been used on
 * a result.
 *) resultSet
  (#
     fetchFirst:
     (* Fetches the first tuple in the resultSet.
      * Advances current tuple, if possible.
      * Current will be EOT if the resultSet is empty *)
       (# result: ^tuple
       ...
       exit result[]
       #);
     fetchLast:
     (* Fetches the last tuple in the resultSet.
      * Current will be EOT if the resultSet is empty *)
       (# result: ^tuple
       ...
       exit result[]
       #);
     fetchNext:
     (* Fetch next tuple in the resultSet.
      * Advance current tuple.
      * Equivalent to 'fetch''
      *)
       (# result: ^tuple
       ...
       exit result[]
       #);
     fetchPrior:
     (* Fetch prior tuple in the resultSet
      * Deadvance current tuple.
      *)
       (# result: ^tuple
       ...
       exit result[]
       #);
     fetchRelative:
     (* Fetches tuple number 'increment' counting from the current tuple
      * If increment is > 0, the current tuple is advanced, if increment is < 0,
      * the current tuple is deadvanced
      * Current will be EOT if the operation is not meaningful *)
       (# increment: @integer; result: ^tuple
       enter increment
       ...
       exit result[]
       #);
     fetchAbsolute:
     (* Positions current at absolute position 'index'
      * If index > 0 then index is counted from start.
      * If index < 0 then index is counted from end,
      *)
       (# index: @integer; result: ^tuple
       enter index
       ...
       exit result[]
```

```
    #);
  scanReverse:
  (* Like 'scan' except that tuples are scanned in reverse order *)
    (#
       current: ^tuple;
       varNotDeclared:<
       (* Raised if variable used in the entered
        * pattern was not found *) exception
         (# name: ^text
         enter name[]
         ...
         #);
       columnNotFound:<
       (* Raised if this is a named scan and
        * named column was not found *) exception
         (# name: ^text
         enter name[]
         ...
         #);
       pattern: ^text
    enter pattern[]
    ...
    #);
  scrollingResultSetException:< BetaDBCException;
  scrollingResultSetNotification:< BetaDBCNotification
  #);
scrollingCursorsSupported: booleanValue
(* Returns true if this(connection) supports scrolling cursors *)
  (#  ... #)
```

Tables Interface

```
ORIGIN 'betadbc';
BODY 'private/tablesbody';
-- connectionLib: Attributes --
table: (* A table in a relational database *)
  (# CatalogName,SchemaName,TableName,TableType: ^text #);
scanTables:
(* Scans the tables in this(connection)
 * invoking INNER for each table found *)
  (# current: ^table ... #)
```

Tables Interface

' Mjłlner Informatics

Transactions Interface

```
ORIGIN 'betadbc';
BODY 'private/transactionsbody';
-- connectionLib: Attributes --
transactionsSupported:
(* Returns true iff this(connection) supports transactions *)
  (# isSupported: @boolean
  ...
  exit isSupported
  #);
autoCommitMode:
(* In autoCommitMode, every database operation is a transaction
 * that is committed when performed. Disabling autoCommitMode is
 * only meaningful when this(connection) supports transactions.
 * The default in BetaDBC is autoCommitMode
 *)
  (# autoCommit: @boolean
  enter (#  enter autoCommit ... #)
  exit
    (#
    ...
    exit
    autoCommit
    #)
  #);
readUncommitted: (* Allows dirty reads, nonrepeatable reads, and phantoms *)
 integerValue (#  ... #);
readCommitted:
(* Disallows dirty reads, but allows nonrepeatable reads and phantoms *)
 integerValue (#  ... #);
repeatableRead:
(* Disallows dirty reads and nonrepeatable reads, but allows phantoms *)
 integerValue (#  ... #);
serializable:
(* Disallows dirty reads, nonrepeatable reads, and phantoms *) integerValue
  (#  ... #);
transactionLevelSupported: booleanValue
(* Check whether level is supported by this(connection) *)
  (# level: (* One of the above defined transaction levels *) @integer
  enter level
  ...
  #);
transactionLevel:
(* Sets the transaction level of the current transaction in
 * this(connection). Serializable is the default
 *)
  (# level: (* One of the above defined transaction levels *) @integer
  enter (#  enter level ... #)
  exit
    (#
    ...
    exit level
    #)
  #);
commit:
(* Commits all transactions pertaining to this connection.
 * Use transactionsSupported to check whether this is meaningful *)
  (#  ... #);
rollBack:
(* Rolls back all transactions pertaining to this(connection)
 * Use transactionsSupported to check whether this is meaningful.
 *) (#  ... #)
```

Transactions Interface