Tutorial 7: Embedded SQL

By Chaofa Gao

Tables used in this note:

Sailors(sid: integer, sname: string, rating: integer, age: real);
Boats(bid: integer, bname: string, color: string);
Reserves(sid: integer, bid: integer, day: date).

Sailors

Sid	Sname	Rating	Age
22	Dustin	7	45
29	Brutus	1	33
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35
64	Horatio	7	35
71	Zorba	10	16
74	Horatio	9	40
85	Art	3	25.5
95	Bob	3	63.5

Boats

bid	bname	Color
101	Interlake	Blue
102	Interlake	Red
103	Clipper	Green
104	Marine	Red

Reserves

sid	bid	day
22	101	1998-10-10
22	102	1998-10-10
22	103	1998-10-8
22	104	1998-10-7
31	102	1998-11-10
31	103	1998-11-6
31	104	1998-11-12
64	101	1998-9-5
64	102	1998-9-8
74	103	1998-9-8

Figure 1: Instances of Sailors, Boats and Reserves

1. Embedded SQL:

The use of SQL command within a host language program is called **Embedded SQL**.
Details of embedded SQL also depend on the host language. The syntax sometimes varies.

1. SQL statements must be marked clearly;
   e.g. insert a row into Table Sailors based on the values in defined variables. (For C)
   EXEC SQL INSERT INTO Sailors VALUES (:c_sid, :c_sname, :c_rating, :c_age)
   In some script languages, you don't have to mark it by yourself. You can write it like other statement in those languages. However, it is the compiler that helps you to mark it during compiling.

2. Any host language variable used to pass arguments into an SQL command must be declared in SQL.
   e.g. Declare variables to pass arguments (For C)

EXEC SQL BEGIN DECLARE SECTION
long c_sid;
char c_sname[20];
short c_rating;
float c_age;
short c_para;
EXEC SQL BEGIN DECLARE SECTION

c_sid long →int; c_sname→ char(20); c_rating→ smallint; c_age→ real
(These data types vary for different languages and different DBMSs.)

You can **cast** data value appropriately before passing them to or from SQL command.
Variables in host languages ←→ Variables in SQL

3. Some special host language variable must be declared in SQL.
   e.g. SQLCODE, SQLSTATE, and so on.
   SQLCODE is defined to return some negative value when an error condition arises, without specifying further just what error a particular negative integer denotes.
   SQLSTATE associates predefined values with several common error conditions, thereby introducing some uniformity to how errors are reports.
   One of these two variables must be declared.
   For C, you can define SQLCODE as long, or SQLSTATE as char[6].

   Here, we assume that SQLSTATE is declared. The variable should be checked for errors and exceptions after each embedded SQL statement. Use WHENEVER to do so.
   EXEC SQL WHENEVER [SQLERROR|NOT FOUND][CONTINUE|GOTO stmt]
   If SQLERROR is specified and the value of SQLSTATE indicates an exception, control is transferred to *stmt*, which is presumably responsible for error/exception handling. Control is also transferred to *stmt* if NOT FOUND is specified and the value of SQLSTATE is 02000, which denotes NOT DATA.

4. Set-oriented:
   SQL commands deal with tables, while the interface to the host language is constrained to **one row at a time**.
   e.g. EXEC SQL SELECT S.sname, S.age
   INTO :c_sname, :c_age
   FROM Sailors S
   WHERE S.sid = :c_sid;
   Given a *c_sid* (e.g. 29), this statement will return this sailor's name and age to variable *c_sname* (e.g. Brutus) and *c_age* (e.g. 33) if this *c_sid* exists.

e.g. EXEC SQL SELECT S.sname, S.age
INTO :c_sname, :c_age
FROM Sailors S
WHERE S.rating > :c_para;

Given a *c_para* (e.g. 5), the SQL command may return more than one row. But the INTO clause is not adequate to support collections of rows. Host variables *c_sname* and *c_age* may just return the first row of the result set.

The solution is to use a cursor. The solution is to essentially provide a mechanism that allows us to retrieve rows one at a time from a relation.

2. Cursor:

We can declare a cursor on any relation or on any SQL query (every query returns a set of rows.)
Once a cursor is **declare**d, we can

1. **open** it (which positions the cursor just before the first row);
2. **fetch** the next row;
3. **move** the cursor (to the next row, to the row after the next *n*, to the first row, or to the previous row etc. by specifying additional parameters for the FETCH command);
4. or **close** the cursor.

```
DECLARE sinfo CURSOR FOR
SELECT S.sname, S.age
FROM Sailors S
WHERE S.rating > :c_para;

OPEN sinfo;

FETCH sinfo INTO :c_sname, :c_age;
Do While (SQLSTATE <> '02000')
    //deal with the current row
    ……
    FETCH sinfo INTO :c_sname, :c_age;
End While

CLOSE sinfo;
```

Look at the special variable SQLCODE or SQLSTATE to find if the FETCH statement positions the cursor after the last row. SQLSTATE is set to the value 02000, which denotes NO DATA. (The syntax varies for different host language.)

Properties of Cursors

The general formal of a cursor declaration is:

DECLARE *cursorname* [INSENSIVE][SCROLL] CURSOR FOR

some query

[ORDER BY order-item-list]

[FOR READ ONLY | FOR UPDATE]

A cursor can be declared to be **a read-only cursor**, or,

if it is a cursor on a base relation or an updateable view, to be **an updateable cursor**.

e. g. if *sinfo* is declared as an updateable cursor and is open, you can modify the *rating* value of the row currently pointed to by cursor *sinfo*.

```
UPDATE Sailors S
SET S.rating = S.rating – 1
WHERE CURRENT of sinfo;
```

You can delete this row by executing the next statement:

```
DELETE Sailors S
WHERE CURRENT of sinfo;
```

A cursor is updateable by default unless it is a scrollable or insensitive cursor, in which case it is read-only by default.

SCROLL, the cursor is scrollable, which means that variants of the FETCH command can be used to position the cursor in very flexible ways;

INSENSTIVE will disregard change to data of other transactions after the cursor is opened.

```
/*****************************************************
**   An example for updateable cursor
*****************************************************/
#define   CHECKERR(CE_STR)    if (sqlca.sqlcode != 0) {printf("%s failed. Reason %ld\n", CE_STR, sqlca.sqlcode); exit(1); }

EXEC SQL DECLARE c1 CURSOR FOR
        SELECT name, dept FROM staff WHERE job='Mgr'
        FOR UPDATE OF job;
```

```
EXEC SQL OPEN c1;
CHECKERR ("OPEN CURSOR");

do {
    EXEC SQL FETCH c1 INTO :pname, :dept;
    if (SQLCODE != 0) break;

    if (dept > 40) {
        printf( "%-10.10s in dept. %2d will be demoted to Clerk\n",
            pname, dept );
        EXEC SQL UPDATE staff SET job = 'Clerk'
            WHERE CURRENT OF c1;
        CHECKERR ("UPDATE STAFF");
    } else {
        printf ("%-10.10s in dept. %2d will be DELETED!\n",
            pname, dept);
        EXEC SQL DELETE FROM staff WHERE CURRENT OF c1;
        CHECKERR ("DELETE");
    } /* endif */
} while ( 1 );

EXEC SQL CLOSE c1;
CHECKERR ("CLOSE CURSOR");
```

Note: before you compile these programs, you need to create the following table and insert some tuples in it.

```
--*************
create table video(video_id int not null, \
                   video_title varchar(30), \
                   director varchar(20), \
                   primary key(video_id))

insert into video values(100,'Titanic','John')
insert into video values(200,'Mask','Mary')
insert into video values(300,'Mary had a little lamb','Jim')
--*************
```

```
/*********************************************************
** A sample program that demonstrates the use of Static embedded SQL.
*********************************************************/
#include <stdio.h>

/* sqlca: is the sql communications area. All error codes are returned from db2 in that structure which is filled each time an interaction with db2 takes place. */

EXEC SQL INCLUDE SQLCA;          /* SQL communication area structure */

EXEC SQL BEGIN DECLARE SECTION;  /* declare host variables      */
  char db_name[8];               /* database name                */
  char video_title[30];          /* title of the video           */
  short video_id;                /* serial number                */
  char director[20];             /* director name                */
EXEC SQL END DECLARE SECTION;

/* These lines are redundant here because the default action is to continue. They just show the kind of errors that could arise and one way to control them.*/

EXEC SQL WHENEVER SQLWARNING CONTINUE;    /* sqlca.sqlcode > 0       */
EXEC SQL WHENEVER SQLERROR CONTINUE;      /* sqlca.sqlcode < 0       */
EXEC SQL WHENEVER NOT FOUND CONTINUE;     /* sqlca.sqlcode = 100     */
                                          /* sqlca.sqlcode = 0 (no error) */

void main() {

strcpy(db_name, "csc343h");
```

```
/* C variables are preceded by a colon when they are passed to DB2 */

EXEC SQL CONNECT TO :db_name;

if (sqlca.sqlcode != 0) {
        printf("Connect failed!: reason %ld\n", sqlca.sqlcode);
        exit(1);
}

EXEC SQL DECLARE c1 CURSOR FOR
        SELECT video_title
        FROM video;

EXEC SQL OPEN c1;   /* you have to open the cursor in order to get tuples back */

do {
   /* fetch tuples from the cursor. */
   EXEC SQL FETCH c1 into :video_title;
   if (SQLCODE != 0) break;        /* SQLCODE refers to sqlca.sqlcode */

   /* host variables should have ':' prefix when they are used in DB2 commands */
   printf("%s\n",video_title);
} while (1);

EXEC SQL CLOSE c1;
EXEC SQL CONNECT RESET;
}
/*************************************************************
** A sample program that demonstrates the use of Dynamic embedded SQL.
*************************************************************/
#include <stdio.h>

EXEC SQL INCLUDE SQLCA;

/* The next statement is used in dynamic SQL programs where the output (the number of columns or
their types) is not fixed in the compile time. Here we don't need it, since the output is fixed. */
/* EXEC SQL INCLUDE SQLDA;                          */

EXEC SQL BEGIN DECLARE SECTION;   /* declare host variables       */
  char    db_name[8];
```

```
  char    qstring[100];                   /* buffer for SQL query          */
  char    video_title[30];
  short video_id;
  char    director[20];
EXEC SQL END DECLARE SECTION;

void main() {

strcpy(db_name, "csc343h");

EXEC SQL CONNECT TO :db_name;

if (SQLCODE != 0) {
            printf("Connect failed!: reason %ld\n", sqlca.sqlcode);
            exit(1);
}

/* construct a query */
strcpy(qstring, "SELECT video_title FROM video");

/* Prepare the query in the run time */
EXEC SQL PREPARE Q1 FROM :qstring;

if (SQLCODE != 0) {
            printf("PREPARE failed!: reason %ld\n", sqlca.sqlcode);
            exit(1);
}

EXEC SQL DECLARE c1 CURSOR FOR Q1;
EXEC SQL OPEN c1;

do {
   EXEC SQL FETCH c1 into :video_title;
   if (SQLCODE != 0) break;

   printf("%s\n",video_title);
} while (1);

EXEC SQL CLOSE c1;
EXEC SQL CONNECT RESET;
}
```

```
# for this to work, you must put
#                 setenv LD_LIBRARY_PATH          /usr/IBMdb2/V7.1/lib
# in you .cshrc
##
# A makefile to compile embeded SQL C programs
# input: sta-select.sqc (a static SQL c program) and
#               dyn-select.sqc (a dynamic SQL c program)
# output: dyn-select    (executable codes)
#

DB2DIR = /usr/IBMdb2/V7.1
Cflags = -I $(DB2DIR)/include \
                -L $(DB2DIR)/lib \
                -ldb2

all: dyn-select

# The first thing is to connect to the database and precompile the programs.
# This generates C programs where all SQL commands (except dynamic SQL
# queries inside dyn-select.sqc) are replaced by subroutine calls.
#

dyn-select.c: dyn-select.sqc
        db2 connect to csc343h
        db2 prep dyn-select.sqc
        db2 connect reset

# The next step is to compile the C programs and
# generate the executable codes.

sta-select: sta-select.c
        gcc -Wall $(Cflags) -o sta-select sta-select.c

dyn-select: dyn-select.c
        gcc -Wall $(Cflags) -o dyn-select dyn-select.c
```

```
*********************************************
** Using SQLSTATE
*********************************************

EXEC SQL CONNECT TO :db_name;

char aux[6];
strcpy(aux,"\0");
strcpy(aux,sqlca.sqlstate);

if (strcmp(aux,"08004")==0 || strcmp(aux,"08002")==0 || strcmp(aux,"08003")==0)
{
        printf("Connect failed!"    );
        exit(1);
}
```