SQL from Applications

UVic C SC 370

Dr. Daniel M. German
Department of Computer Science

University of Victoria

June 4, 2003 Version: 1.1.0

Overview

- Embedded SQL
- JDBC
- Stored Procedures

Accesing data from an application

- Most of the time, an application will be the interface between the database and the user
- Almost any programming language can be used to do it
  - C
  - Perl
  - Java
  - PHP
  - ASP
  - Tcl
  - Python
  - You name it!

JDBC

- *Java DataBase Connectivity*
- JDBC allows database independence at the run-time level
- One program can use several databases at the same time
- A **driver** is responsible for the interaction with a particular DBMS
- Drivers are loaded dynamically
- *IMHO, JDBC is the way to go when speed is not an issue*

Example

```
import java.sql.*;

class sailors
{
    static public void main(String[] args)
    {
        try {
            // Prepare driver
            Class.forName("org.postgresql.Driver");
            // connect to the database
            Connection connection = DriverManager.getConnection
                ("jdbc:postgresql:sail3", "dmg", "password");

            String sqlCommand = "select sname from Sailors order by sname";
            Statement statement = connection.createStatement();
            ResultSet result = statement.executeQuery(sqlCommand);
            String sName;

            while (result.next()) {
                sName = result.getString(1);
                System.out.println(sName);
            }
            connection.close();
        }
        catch (java.lang.Exception ex) {
            System.out.println("Connect or execute query exception: " + ex);
            ex.printStackTrace();
        }
    }
}
```


Embedded SQL in C

- The idea is to embed SQL into an application written in C
- A preprocessor takes care of translating the SQL into host language primitives and function calls:
- Example: select all names of sailors

Example

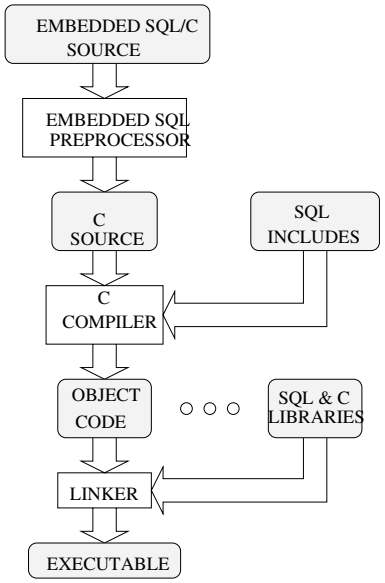
```
/*
 *  embedded C sample program
 */
#include <stdio.h>

void My_Exit(void);

EXEC SQL INCLUDE sqlca;

/* Whenever there is an error call error handler */
EXEC SQL WHENEVER SQLERROR DO My_Exit();

int main(int numParms, char* parms[])
{

EXEC SQL BEGIN DECLARE SECTION;
    char *c_sname = NULL;       /* holds value returned by query */
    char  c_query_string[256];  /* holds constructed SQL query   */
    char  c_dbName[40], c_userName[20], c_password[20];
    int   c_rating, c_maxRating;
EXEC SQL END DECLARE SECTION;

    int         rating;

    if (numParms != 3) {
       fprintf(stderr, "Usage %s <username> <password>\n", parms[0]);
       exit(1);
    }
    strcpy(c_dbName, "csc370public@postgresql.csc.uvic.ca");
    strcpy(c_userName,  parms[1]);
```



```
    strcpy(c_password, parms[2]);

    /* connect to the database */

    EXEC SQL CONNECT TO :c_dbName USER :c_userName USING :c_password;

    /* get a singleton */
    EXEC SQL SELECT Max(rating) INTO :c_maxRating FROM sailors;

    /* USING CURSORS */
    rating = 3;
    sprintf(c_query_string,        /* create an SQL query string */
                "SELECT sname, rating \
                 FROM sailors WHERE rating >= %d\
                 ORDER BY rating", rating);

    /* Prepare query */
    EXEC SQL PREPARE s_sailorName FROM :c_query_string;

    /* DECLARE a cursor for that query*/
    EXEC SQL DECLARE cursorSailor CURSOR FOR s_sailorName;
    EXEC SQL OPEN cursorSailor;           /* send the query */

    EXEC SQL WHENEVER NOT FOUND DO BREAK; /* Break out of the loop where
                                             no more rows */
    while (1) {
        EXEC SQL FETCH IN cursorSailor INTO :c_sname, :c_rating;
        printf("%s %-3d (max %d)\n", c_sname, c_rating, c_maxRating);
    }
    EXEC SQL CLOSE cursorSailor;  /* CLOSE the cursor */
    EXEC SQL COMMIT;
    EXEC SQL DISCONNECT;          /* disconnect from the database */
    return 0;
}
```

```
/*
 *  Error handler: print error and die
 */
void My_Exit(void)
{
   fprintf(stderr, "Error in SQL operation: %s\n", sqlca.sqlerrm.sqlerrmc);
   exit(1);
}
```


Embedded SQL, how it works



A Makefile for the program

```
PSQLLIB=/public/lib
PSQLINC=/public/include

default: sailors

sailors: sailors.c
        gcc -g -I ${PSQLINC} -o sailors sailors.c -L ${PSQLLIB} -lecpg

sailors.c: sailors.csql
        ecpg $<
```


Declaring Variables

- SQL programs can refer to variables defined in the host program.
- These variables must be declared between the commands `EXEC SQL BEGIN DECLARE SECTION` and `EXEC SQL END DECLARE SECTION`
- These declarations look like normal C declarations
- The use of **c_** as prefix of the var name is a convention to clarify they are host variables

```
EXEC SQL BEGIN DECLARE SECTION;
    char *c_sname = NULL;       /* holds value returned by query */
    char  c_query_string[256];  /* holds constructed SQL query   */
    char  c_dbName[40], c_userName[20], c_password[20];
    int   c_rating, c_maxRating;
EXEC SQL END DECLARE SECTION;
```

Embedding SQL statements

- All SQL statements should be clearly delimited. In C you use `EXEC SQL`
- An SQL statement can be used as a regular statement in C
- Any time a SQL statement uses a host variable it should be prefixed with **:**
- A semicolon **;** ends the statement
- Example:

```
EXEC SQL
INSERT INTO Sailors VALUES (:c_sname, :c_rating, c_age);
```


Error Handling

- To simplify error handling, use the WHENEVER

```
EXEC SQL WHENEVER [NOT FOUND | SQLERROR ]
      [CONTINUE | DO c-statement | GOTO stmt];
```

- `NOT FOUND`: When the last record of a set has been read, do...
- `SQLERROR`: Whenever there is an error, do...
- Examples:

```
EXEC SQL WHENEVER SQLERROR DO My_Exit();
EXEC SQL WHENEVER NOT FOUND DO BREAK;
```


Cursors

- C does not cleanly support sets.
- Furthermore, the result of a query might have more rows than can fit in memory.
- Cursors allow us to retrieve, from a result set, one row at a time.
- Abstraction:
  - You `DECLARE` a cursor: prepares the query to be executed
  - You `OPEN` a cursor: executes the query and positions the cursor **before** the first row
  - You `FETCH` from a cursor: positions the cursor in the **next** row
  - You `CLOSE` the cursor when you are done with it.

Cursors...

- Example:

```
rating = 3;
sprintf(c_query_string,        /* create an SQL query string */
            "SELECT sname, rating \
             FROM sailors WHERE rating >= %d\
             ORDER BY rating", rating);

/* Prepare query */
EXEC SQL PREPARE s_sailorName FROM :c_query_string;


EXEC SQL DECLARE cursorSailor CURSOR FOR s_sailorName;
EXEC SQL OPEN cursorSailor;           /* send the query */
EXEC SQL WHENEVER NOT FOUND DO BREAK; /* Break out of the loop where
                                         no more rows */
while (1) {
    EXEC SQL FETCH IN cursorSailor INTO :c_sname, :c_rating;
    printf("%s %-3d (max %d)\n", c_sname, c_rating, c_maxRating);
}
EXEC SQL CLOSE cursorSailor;  /* CLOSE the cursor */
```

Singletons

- When the query returns a singleton, we can avoid using a cursor:

```
EXEC SQL SELECT Max(rating) INTO :c_maxRating FROM sailors;

EXEC SQL SELECT S.sname, S.age
   INTO :c_sname, :c_age
   FROM sailors
   WHERE S.sid = :c_sid;
```


Cursors...

- Form of a cursor declaration:

```
DECLARE cursorname [INSENSITIVE] [SCROLL] CURSOR
    [WITH HOLD]
    FOR somequery
    [FOR READ ONLY | FOR UPDATE]
```

- A cursor by default is `FOR UPDATE` and allows the following (where sinfo is a cursor):

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

- A `SCROLL` cursor allows you to move around in the result set:

```
EXEC SQL MOVE -3 FROM sinfo;
```

- A `INSENSITIVE` cursor makes an entire copy of the result set before the first fetch.
- A `WITH HOLD` cursor allows to create a transaction per row, instead of a transaction for the entire duration of the cursor.

More examples

From the postgresql documentation:

```
EXEC SQL CREATE TABLE foo (number int4, ascii char(16));
EXEC SQL CREATE UNIQUE index num1 on foo(number);
EXEC SQL COMMIT;

EXEC SQL INSERT INTO foo (number, ascii) VALUES (9999, 'doodad');
EXEC SQL COMMIT;

EXEC SQL DELETE FROM foo WHERE number = 9999;
EXEC SQL COMMIT;

EXEC SQL UPDATE foo
    SET ascii = 'foobar'
    WHERE number = 9999;
EXEC SQL COMMIT;
```


Extending the DBMS

- You can extend the functionality of the database using **user defined functions** (UDFs) and **stored procedures** (SPs)
- UDFs return a value, SPs do not necessarily
- They can be written in virtually any host language, including SQL
- ok, ok, postgresql does not support SPs in SQL

Example

- This query counts the number of sailors for a given rating

```
CREATE FUNCTION CountSailorsWithRating(integer) RETURNS BIGINT AS '
   SELECT Count(*) from Sailors where rating = $1
' LANGUAGE SQL;
```

- It is used as any other library function:

```
SELECT DISTINCT  rating,
                 CountSailorsWithRating(rating) AS COUNT
      FROM sailors ORDER BY rating DESC;
```

Same as:

```
select rating, count(*) from sailors group by rating
  ORDER BY rating DESC;
```


UDFs and SPs

- SPs and UDFs can be executed in the DBMS space, saving time and resources
- Stored procedures are good Soft. Eng.
- UDFs can be very powerful, but can become very inefficient (see example in previous page) *be careful in their use*

Triggers

- **Triggers** are procedures that can be automatically invoked in response to a change in the database
- A trigger contains three parts:
  - **Event**: What change in the database will activate the trigger
  - **Condition** (not always supported): A query to test if the trigger should be activated
  - **Action**: What to do when the trigger is **activated** and the **condition** is true
- Example:

```
CREATE TRIGGER SaveOldSailors BEFORE DELETE ON Sailors
   FOR EACH ROW
   EXECUTE InsertIntoSavedSailors(old.sid, old.sname);
```


Triggers...

- General form (in postgresql):

```
CREATE TRIGGER trigger [ BEFORE | AFTER ]
                       [ INSERT | DELETE | UPDATE [ OR ... ] ]
    ON relation FOR EACH [ ROW | STATEMENT ]
    EXECUTE PROCEDURE procedure(args);
```

- `BEFORE | AFTER`: when is it called?
- `INSERT | DELETE | UPDATE`: during what operation(s)?
- `ROW | STATEMENT`: call it once per row or once per statement? (for example, UPDATE ... WHERE is one statement that can modify zero or more rows)