Embedded / Dynamic SQL

Overview

- Database Programming
- Impedance Mismatch
- Embedded SQL
  - Preliminary details
  - Points arising – Cursors
  - Operations involving Cursors
  - Dynamic SQL
- Database Programming with Function Calls
  - SQL/CLI
  - JDBC

Database Programming

- Objective:
  - To access a database from an application program (as opposed to interactive interfaces)
- Why?
  - An interactive interface is convenient but not sufficient
    - A majority of database operations are made through application programs (increasingly through web applications)

Database Programming Approaches

- Embedded commands:
  - Database commands are embedded in a general-purpose programming language
  - Database statements are identified by prefix EXEC SQL
  - A pre-compiler scans the source program to identify database statements
  - They are replaced in the program by function calls to the DBMS-generated code

Database Programming Approaches

- Library of database functions:
  - Available to the host language for database calls; known as an *API* (Application Program Interface)
  - Functions to connect to database, execute a query, execute an update.
- A brand new, full-fledged language
  - Minimizes impedance mismatch

Impedance Mismatch

- Differences between database model and programming language model – issue
- Embedded commands:
  - type mismatch and incompatibilities; requires a new binding for each language.
  - set vs. record-at-a-time processing.
    - need special iterators to loop over query results and manipulate individual values.

Database Programming

- Client program *opens a connection* to the database server
- Client program *submits queries to and/or updates* the Database
- When database access is no longer needed, client program *closes (terminates) the connection*

Embedded SQL

Embedded SQL

- SQL statements can be embedded in a general-purpose *host* programming language such as COBOL, C/C++, Java.
- An embedded SQL statement is distinguished from the host language statements by enclosing it between **EXEC SQL** or **EXEC SQL BEGIN** and a matching **END-EXEC** or **EXEC SQL END** (or a semicolon ;)
  - Syntax may vary with language.
  - *Shared variables* (used in both languages) usually prefixed with a colon (:) in SQL.

Declaring variables

- Variables inside **DECLARE** are shared and can appear (while prefixed by a colon) in SQL statements
- **SQLCODE/SQLSTATE** is used to communicate errors/exceptions between the database and the program

```
0)    int loop ;
1)    EXEC SQL BEGIN DECLARE SECTION ;
2)    varchar dname [16], fname [16], lname [16], address [31] ;
3)    char ssn [10], bdate [11], sex [2], minit [2] ;
4)    float salary, raise ;
5)    int dno, dnumber ;
6)    int SQLCODE ; char SQLSTATE [6] ;
7)    EXEC SQL END DECLARE SECTION ;
```

Connecting to Database

- Connection (multiple connections are possible but only one is active)

  CONNECT TO server-name **AS** connection-name

  AUTHORIZATION user-account name and pwd**;**
- Change from an active connection to another one

  SET CONNECTION connection-name**;**
- Disconnection

  DISCONNECT connection-name**;**

Preliminary details

- Embedded SQL statements are prefixed by EXEC SQL, and terminated by a special terminator symbol.
- SQL statements can include references to host variables; such references must include a colon prefix.
- All host variables must be declared within an embedded SQL declare section.
- After the execution of any SQL statement, a status code is returned to the program in a host variable called SQLSTATE.

Example – Embedded SQL

- SQLCODE=0 – indicates statement was executed successfully
- SQLCODE<0 – indicates some error has occurred
- SQLCODE>0 – indicates no data found

```
//Program Segment E1:
0)    loop = 1 ;
1)    while (loop) {
2)      prompt("Enter a Social Security Number: ", ssn) ;
3)      EXEC SQL
4)        select FNAME, MINIT, LNAME, ADDRESS, SALARY
5)        into :fname, :minit, :lname, :address, :salary
6)        from EMPLOYEE where SSN = :ssn ;
7)      if (SQLCODE == 0) printf(fname, minit, lname, address, salary)
8)        else printf("Social Security Number does not exist: ", ssn) ;
9)      prompt("More Social Security Numbers (enter 1 for Yes, 0 for No): ", loop) ;
10)     }
```

Points arising

- Retrieval operations requires special treatment since it retrieves many rows, not just one.
- Host languages are generally not equipped to handle the retrieval of more than one row at a time.
- To bridge the gap between set-level retrieval capabilities of SQL and row-level retrieval capabilities of host – **cursor** are used.
- Cursor – a logical pointer (in application) pointing to each of the rows thereby providing addressability to those rows one at a time.

Operations Not Involving Cursors

- The DML statements that do not need cursors are as follows:

Singleton SELECT
INSERT
DELETE
UPDATE

Singleton SELECT

- SELECT expression that evaluates to a table containing at most one row:

```
EXEC SQL SELECT STATUS, CITY
            INTO :RANK, :TOWN
            FROM SUPPLIERS
            WHERE SUPPLIER_NUMBER = :GIVEN SUPPLIER_NUMBER
```

INSERT, UPDATE

- Insert a new part:

```
EXEC SQL INSERT
        INTO PARTS (PART_NUMBER, PART_NAME, WEIGHT)
        VALUES ( :PART_NUMBER, :PART_NAME, :PWT );
```

- Increase the status of all Bombay suppliers by the amount given by host variable RAISE.

```
EXEC SQL UPDATE SUPPLIERS
        SET STATUS = STATUS + :RAISE
        WHERE CITY = 'Bombay' ;
```

Operations Involving Cursors

- Cursor – a mechanism for accessing the rows in the set one by one.
- Cursor is declared by means of DECLARE CURSOR.
- The table expression is evaluated when the cursor is opened.

```
EXEC SQL DECLARE <cursor name> CURSOR
        FOR [ table expression ]                    /*define the cursor */
```

```
EXEC SQL DECLARE X CURSOR FOR
        SELECT S.SUPPLIER_NUMBER, S.SUPPLIER_NAME, S.STATUS
        FROM SUPPLIERS S
        WHERE S.CITY = :Y ;
```

Operations Involving Cursors

- Three executable statements are provided to operate on cursors: OPEN, FETCH, and CLOSE.

EXEC SQL OPEN <cursor name> ;

- opens the specified cursor.
- the table expression associated with the cursor is evaluated.
- a set of rows becomes the current **active set** for the cursor.

Operations Involving Cursors

- Three executable statements are provided to operate on cursors: OPEN, FETCH, and CLOSE.

EXEC SQL FETCH <cursor name> INTO <host variable reference>;

- advances the specified cursor to the next row in the active set

- then assigns the *i* th value to the *i* th host variable

EXEC SQL CLOSE <cursor name>;

closes the specified cursor. The cursor now has no current active set.

Operations Involving Cursors

- FETCH appears inside the loop since there will be many rows in the result set.
- INTO clause assigns the retrieved values to host variables.

```
EXEC SQL DECLARE X CURSOR FOR
        SELECT S.SUPPLIER_NUMBER, S.SUPPLIER_NAME, S.STATUS
        FROM SUPPLIERS S
        WHERE S.CITY = :Y ;

EXEC SQL OPEN X;
    DO for all SUPPLIERS rows accessible via X;
        EXEC SQL FETCH X INTO :SUPPLIER_NUMBER, :SUPPLIER_NAME, :STATUS
        .....                                          /* fetch next supplier*/
    END;
EXEC SQL CLOSE X;
```

Example – with cursor

- Raise the salary of each employee for the given department name.

```
//Program Segment E2:
0)   prompt("Enter the Department Name: ", dname) ;
1)   EXEC SQL
2)       select DNUMBER into :dnumber
3)       from DEPARTMENT where DNAME = :dname ;
4)   EXEC SQL DECLARE EMP CURSOR FOR
5)       select SSN, FNAME, MINIT, LNAME, SALARY
6)       from EMPLOYEE where DNO = :dnumber
7)       FOR UPDATE OF SALARY ;
8)   EXEC SQL OPEN EMP ;
9)   EXEC SQL FETCH from EMP into :ssn, :fname, :minit, :lname, :salary ;
10)  while (SQLCODE == 0) {
11)      printf("Employee name is:", fname, minit, lname)
12)      prompt("Enter the raise amount: ", raise) ;
13)      EXEC SQL
14)          update EMPLOYEE
15)          set SALARY = SALARY + :raise
16)          where CURRENT OF EMP ;
17)      EXEC SQL FETCH from EMP into :ssn, :fname, :minit, :lname, :salary ;
18)      }
19)  EXEC SQL CLOSE EMP ;
```

Dynamic SQL

- Dynamic SQL is part of Embedded SQL.
- In embedded SQL – queries were part of host program source code
- Dynamic statements are compiled ahead of time : main purpose is to execute SQL that are constructed at run time.
- Dynamic update is relatively simple; dynamic query can be complex
  - because the type and number of retrieved attributes are unknown at compile time

Dynamic SQL

- The two principle dynamic statements are: PREPARE and EXECUTE.

```
DCL SQLSOURCE CHAR VARYING (65000);

SQLSOURCE = 'DELETE FROM SHIPMENTS WHERE QUANTITY< QUANTITY(300)';
EXEC SQL PREPARE SQLPREPPED FROM :SQLSOURCE;
EXEC SQL EXECUTE SQLPREPPED;
```

```
//Program Segment E3:
0)    EXEC SQL BEGIN DECLARE SECTION ;
1)    varchar sqlupdatestring [256] ;
2)    EXEC SQL END DECLARE SECTION ;
      …
3)    prompt("Enter the Update Command: ", sqlupdatestring) ;
4)    EXEC SQL PREPARE sqlcommand FROM :sqlupdatestring ;
5)    EXEC SQL EXECUTE sqlcommand ;
      …
```

Database Programming with Function Calls: SQL/CLI, JDBC

Dynamic SQL

- Embedded SQL provides static database programming
- API: Dynamic database programming with a library of functions
- Advantage:
  - No preprocessor needed (thus more flexible)
- Disadvantage:
  - SQL syntax checks to be done at run-time

SQL Call-Level Interfaces

- A part of the SQL standard
- Provides easy access to several databases within the same program – *DBMS-independent*
- Certain libraries (e.g., **sqlcli.h** for C) have to be installed and available
- SQL statements are dynamically created and passed as string parameters in the calls

Components of SQL/CLI

- *Environment record*:
  - Keeps track of database connections
- *Connection record*:
  - Keep tracks of info needed for a particular connection
- *Statement record*:
  - Keeps track of info needed for one SQL statement
- *Description record*:
  - Keeps track of tuples

Steps in C and SQL/CLI programming

1. *Load SQL/CLI libraries*
2. Declare record handle variables for the above components (called: **SQLHENV, SQLHDBC, SQLHSTMT, SQLHDEC**)
3. Set up an environment record using **SQLAllocHandle**
4. Set up a connection record using **SQLAllocHandle**
5. Set up a statement record using **SQLAllocHandle**
6. Prepare a statement using SQL/CLI function **SQLPrepare**
7. Bound parameters to program variables
8. Execute SQL statement via **SQLExecute**
9. Bound query columns to a C variable via **SQLBindCol**
10. Use **SQLFetch** to retrieve column vals into C variables

C with SQL/CLI

SQLAllocHandle (<handle_type>,<container-handle>,<ptr-handle>)

```
//Program CLI1:
0)   #include sqlcli.h ;
1)   void printSal() {
2)   SQLHSTMT stmt1 ;
3)   SQLHDBC con1 ;
4)   SQLHENV env1 ;
5)   SQLRETURN ret1, ret2, ret3, ret4 ;
6)   ret1 = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &env1) ;
7)   if (!ret1) ret2 = SQLAllocHandle(SQL_HANDLE_DBC, env1, &con1) else exit ;
8)   if (!ret2) ret3 = SQLConnect(con1, "dbs", SQL_NTS, "js", SQL_NTS, "xyz", SQL_NTS)
else exit ;
9)   if (!ret3) ret4 = SQLAllocHandle(SQL_HANDLE_STMT, con1, &stmt1) else exit ;
10)  SQLPrepare(stmt1, "select LNAME, SALARY from EMPLOYEE where SSN = ?", SQL_NTS) ;
11)  prompt("Enter a Social Security Number: ", ssn) ;
12)  SQLBindParameter(stmt1, 1, SQL_CHAR, &ssn, 9, &fetchlen1) ;
13)  ret1 = SQLExecute(stmt1) ;
14)  if (!ret1) {
15)      SQLBindCol(stmt1, 1, SQL_CHAR, &lname, 15, &fetchlen1) ;
16)      SQLBindCol(stmt1, 2, SQL_FLOAT, &salary, 4, &fetchlen2) ;
17)      ret2 = SQLFetch(stmt1) ;
18)      if (!ret2) printf(ssn, lname, salary)
19)          else printf("Social Security Number does not exist: ", ssn) ;
20)      }
21)  }
```

JDBC: SQL Function Call for Java

- JDBC: SQL connection function calls for Java programming
- A Java program with JDBC functions can access any relational DBMS that has a JDBC driver
- JDBC allows a program to connect to several databases (known as *data sources*)

Steps in JDBC Database Access

1. Import JDBC library **(java.sql.*)**
2. Load JDBC driver:

   Class.forname(“oracle.jdbc.driver.OracleDriver”)
3. Define appropriate variables
4. Create a connect object (via **getConnection**)
5. Create a statement object from the **Statement** class:

   1. PreparedStatment 2. CallableStatement
6. Identify statement parameters (designated by ? Marks)
7. Bound parameters to program variables

Steps in JDBC Database Access

8. Execute SQL statement (referenced by an object) via JDBC's **executeQuery**
9. Process query results (returned in an object of type ResultSet) – **ResultSet** is a 2-dimentional table

Java with JDBC

```
//Program JDBC1:
0)    import java.io.* ;
1)    import java.sql.*
      …
2)    class getEmpInfo {
3)       public static void main (String args []) throws SQLException, IOException {
4)       try { Class.forName("oracle.jdbc.driver.OracleDriver")
5)       } catch (ClassNotFoundException x) {
6)          System.out.println ("Driver could not be loaded") ;
7)          }
8)       String dbacct, passwrd, ssn, lname ;
9)       Double salary ;
10)      dbacct = readentry("Enter database account:") ;
11)      passwrd = readentry("Enter pasword:") ;
12)      Connection conn = DriverManager.getConnection
13)         ("jdbc:oracle:oci8:" + dbacct + "/" + passwrd) ;
14)      String stmt1 = "select LNAME, SALARY from EMPLOYEE where SSN = ?" ;
15)      PreparedStatement p = conn.prepareStatement(stmt1) ;
16)      ssn = readentry("Enter a Social Security Number: ") ;
17)      p.clearParameters() ;
18)      p.setString(1, ssn) ;
19)      ResultSet r = p.executeQuery() ;
20)      while (r.next()) {
21)         lname = r.getString(1) ;
22)         salary = r.getDouble(2) ;
23)         system.out.println(lname + salary) ;
24)      }  }
25)   }
```

References

- Fundamentals of Database Systems, Elmasri, Navathe, 5th Ed.
- An introduction to database systems, *CJ. Date*