7 Embedded SQL

Prerequisites for This Section

- **Readings:**
  - **Required:** Connolly and Begg, sections E.1.1 and E.2 (in third edition, sections 21.1.1, 21.2.1, and 21.2.2).
- **Assessments:**
  - Multiple-Choice Quiz 4

Section Objectives

In this section you will learn:

① How SQL statement can be embedded in high-level programming language.

② The differences between static and dynamic embedded SQL.

③ How to write programs that use static embedded SQL statements.

④ How to write programs that use dynamic embedded SQL statements.

Agenda

1. Programmatic SQL
2. Simple Embedded SQL Statements
3. SQL Communication Area
4. Host Language Variables
5. Cursors
6. Dynamic SQL

SQL2 Lacks Computational Completeness

- 1992 SQL standard lacked computational completeness: it contained no flow of control commands such as IF …THEN … ELSE, GO TO, or DO … WHILE.
- SQL contained only definitional and manipulative commands.
- To overcome this and provide more flexibility, SQL allows statements to be **embedded** in a high-level procedural language, as well as being able to enter SQL statements **interactively** at a terminal.

Two Types of Programmatic SQL

① **Embedded SQL** Statements

- SQL statements are embedded directly into the program source code and mixed with the host language statements.

② **Application Programming Interface** (**API**)

- An alternative technique is to provide the programmer with a standard set of functions that can be invoked from the software.

① Embedded SQL

- This approach allows users to write programs that access the database directly.
- A special precompiler modifies the source code to replace SQL statements with calls to DBMS routines. The source code can then be compiled and linked in the normal way.
- Two types of embedded SQL
  - Static embedded SQL
  - Dynamic embedded SQL

① *Embedded SQL*

- **Static embedded SQL**: where the entire SQL statement is known when the program is written
- **Dynamic embedded SQL**: which allows all or part of the SQL statement to be specified at runtime.
- To make the discussions more concrete, we demonstrate the Oracle 9i dialect of SQL embedded in the ‘C’ programming language.

② *Application Programming Interface (API)*

- An API can provide the same functionality as embedded statements and removes the need for any precompilation.
- **ODBC**: the best-known API is the Open Database Connectivity (ODBC) standard. It is built on the 'C' language.
- **JDBC**: the most prominent and mature approach for accessing relational DBMSs from Java appears to be JDBC (Java Database Connectivity).

Agenda

1. Programmatic SQL
2. Simple Embedded SQL Statements
3. SQL Communication Area
4. Host Language Variables
5. Cursors
6. Dynamic SQL

Some Basic Rules

① Embedded SQL statements can continue over more than one line, using the continuation marker of the host language.

② An embedded SQL statement can appear anywhere that an executable host language statement can appear.

Embedded SQL Programming Needs to

① Help precompiler to recognize embedded SQL statements from high-level source code

----- A start identifier and a terminator

② Report runtime errors of SQLs to the application program

----- SQL Communication area

③ Transfer data from the database into the program, and vice versa

----- Host Language Variable

④ Overcome the **impedance mismatch** between high-level programming languages and SQLs

----- Cursor

Start Identifier and Terminator

- **Start identifier**
  - Embedded SQL statements start with an identifier usually the keyword **EXEC SQL**.
- **Terminator**
  - Embedded SQL statements end with a terminator that is dependent on the host language.
  - In ‘C’ language, the terminator is a semicolon **(;)**.

Example of an Embedded SQL Statement

- Create the **Viewing table** interactively:

```
CREATE TABLE Viewing (propertyNo  VARCHAR(5) NOT NULL,
                      clientNo    VARCHAR(5) NOT NULL,
                      viewDate    DATE       NOT NULL,
                      comments    VARCHAR(40) );
```

- The embedded SQL statement:

```
EXEC SQL CREATE TABLE Viewing (propertyNo  VARCHAR(5) NOT NULL,
                               clientNo    VARCHAR(5) NOT NULL,
                               viewDate    DATE       NOT NULL,
                               comments    VARCHAR(40) );
```

Agenda

1. Programmatic SQL
2. Simple Embedded SQL Statements
3. SQL Communication Area
4. Host Language Variables
5. Cursors
6. Dynamic SQL

SQLCA

- The DBMS uses a SQL Communication Area (SQLCA) to report runtime errors to the application program.
- **SQLCA is a data structure** that contains error variables and status indicators.
- An application program can examine the SQLCA to determine the success or failure of each SQL Statement.

Using SQLCA

- At the start of the program we include the line:

  EXEC SQL INCLUDE sqlca;

- This tells the precompiler to include the SQLCA data structure in the program.

- The most important part of SQLCA is the **SQLCODE** variable.

- SQLCODE is set by DBMS as follows:
  - ① =0: statement executed successfully
  - ② <0: an error occurred
  - ③ >0: statement executed successfully, but an exceptional condition occurred

Example of Using SQLCODE

```
EXEC SQL CONNECT  :username IDENTIFIED BY :password
                  USING :db_name;

if (sqlca.sqlcode  < 0) exit (-1);

EXEC SQL CREATE TABLE Viewing (   propertyNo VARCHAR(5) NOT NULL,
                                  clientNo     VARCHAR(5) NOT NULL,
                                  viewDate     DATE       NOT NULL,
                                  comments     VARCHAR(40) );

if  (sqlca.sqlcode >= 0)
        printf ("Creation Successful\n");
else
        printf ("Creation Unsuccessful\n");
```

Example of Using SQLCODE

Every embedded SQL statement can potentially generate an error. Clearly, checking for success after every SQL statement would be quite laborious.

```
EXEC SQL INSERT INTO viewing VALUES ('CR76',
    'PA14', '12-May-2001', 'Not enough space');

if (sqlca.sqlcode < 0) goto error1;

EXEC SQL INSERT INTO viewing VALUES ('CR77',
    'PA14', '13-May-2001', 'Quite like it');

if (sqlca.sqlcode < 0) goto error1;
```

WHENEVER Statement

- The WHENEVER statement is a directive to precompiler to automatically generate code to handle error after every SQL statement.
- The format of the WHENEVER statement is:

  EXEC SQL WHENEVER <condition> <action>

- The WHENEVER statement consists of a **condition** and an **action** to be taken if the condition occurs.

WHENEVER Statement-- Condition

- The **condition** can be one of the following:

① **SQLERROR**

tell the precompiler to generate code to handle error (SQLCODE < 0)

② **SQLWARNING**

tell the precompiler to generate code to handle warning (SQLCODE > 0)

③ **NOT FOUND**

tell the precompiler to generate code to handle a specific warning that a retrieval operation has found no more records.

WHENEVER Statement --- Action

- The **action** can be:
  - ① **CONTINUE**, to ignore the condition and proceed to the next statement
  - ② **DO**, to transfer control to an error handling function.
  - ③ **DO BREAK**, to place an actual "break" statement in the program.
  - ④ **DO CONTINUE**, to place an actual "continue" statement in the program.
  - ⑤ **GOTO** *label*, to transfer control to the specified *label*.
  - ⑥ **STOP**, to rollback all uncommitted work and terminate the program.

Example of WHENEVER Statement

- For example, the WHENEVER statement in the code segment:

EXEC SQL WHENEVER SQLERROR GOTO error1;

EXEC SQL INSERT INTO viewing VALUES ('CR76', 'PA14', '12-May-2001', 'Not enough space');

EXEC SQL INSERT INTO viewing VALUES ('CR77', 'PA14', '13-May-2001', 'Quite like it');

- Would be converted by the precompiler to:

EXEC SQL INSERT INTO viewing VALUES ('CR76', 'PA14', '12-May-2001', 'Not enough space');

if (sqlca.sqlcode < 0) goto error1;

EXEC SQL INSERT INTO viewing VALUES ('CR77', 'PA14', '13-May-2001', 'Quite like it');

if (sqlca.sqlcode < 0) goto error1;

Agenda

1. Programmatic SQL
2. Simple Embedded SQL Statements
3. SQL Communication Area
4. Host Language Variables
5. Cursors
6. Dynamic SQL

Host Language Variable

- A host language variable is a program variable declared in the host language.
- It **can be**
  - either a single variable or a structure .
  - used in embedded SQL statement to transfer data from the database into the program, and vice versa.
  - used within the WHERE clause of SELECT statement.

Declaration of Host Variable

- To use a host language variable in an embedded SQL statement, the variable name is prefixed by a colon (**:**).
- All host variables must be declared to SQL in a **BEGIN DECLARE SECTION … END DECLARE SECTION** block.
- For example:

EXEC SQL BEGIN DECLARE SECTION;

float increment;

EXEC SQL END DECLARE SECTION;

EXEC SQL UPDATE staff SET salary = salary + **: increment**

WHERE staffNo = ‘SL21’;

Indicator Variable

- Problem: most programming languages do not provide support for unknown or missing values. This causes a problem when a **null** has to be inserted or retrieved from a table.
- Solution: **indicator variables**
- The meaning of the **indicator variable** is as follows:

  ① **0**: the associated host variable contains a valid value

  ② **-1**: the associated host variable should be assumed to contain a null

  ③ **>0**: the associated host variable contains a valid value, which may have been rounded or truncated

Example of Indicator Variable

- To set the address column of owner CO21 to NULL

```
EXEC SQL BEGIN DECLARE SECTION;
        char   address[51];
        short  addressInd;
EXEC SQL END DECLARE SECTION;
addressInd = -1;
EXEC SQL UPDATE PrivateOwner
        SET  address = :address : addressInd
        WHERE OwnerNo = 'CO21';
```

Agenda

1. Programmatic SQL
2. Simple Embedded SQL Statements
3. SQL Communication Area
4. Host Language Variables
5. Cursors
6. Dynamic SQL

Single-row Queries

- In embedded SQL, single-row queries are handled by the **singleton select** statement, which has the same format as the SELECT statement interactively used, with an extra INTO clause specifying the names of the host variables to receive the query result.
- For example, to retrieve details of owner CO21:

```
EXEC SQL SELECT   fName, lName, address
                  INTO   :fName, :lName, :address :addressInd
                  FROM   PrivateOwner
                  WHERE ownerNo = 'CO21';
```

Multi-row Queries

- When a database query can return an arbitrary number of rows, embedded SQL uses **cursors** to return the data.
- A cursor allows a host language to access the rows of a query result one at a time.
- A cursor must be **declared** and **opened** before it can be used, and it must be **closed** to deactivate it after it is no longer required.

ISO Standard for Embedded SQL

- **DECLARE** cursor statement:

EXEC SQL DECLARE cursorName CURSOR FOR selectStatement

[FOR {READ ONLY | UPDATE [OF columnnameList]}]

- **OPEN** cursor statement:

EXEC SQL OPEN cursorName

- **FETCH** cursor statement:

EXEC SQL FETCH cursorName INTO [hostVariable [indicatorVariable][, …]]

- **CLOSE** cursor statement:

EXEC SQL CLOSE cursorName

DECLARE Cursor

EXEC SQL DECLARE cursorName CURSOR FOR selectStatement

[FOR {**READ ONLY | UPDATE** [OF columnnameList]}]

- If the table/view identified by a cursor is not updatable, then the cursor is **readonly**; otherwise, the cursor is **updatable**, and the positioned UPDATE and DELETE CURRENT statements can be used.
- Rows can always be inserted directly into the base table . If rows are inserted after the current cursor and the cursor is readonly, the effect of the change is not visible through that cursor before it is closed. If the cursor is updatable, the effect of such changes is implementation-dependent.

Using Cursors to Retrieve Data

- EXEC SQL **DECLARE** propertyCursor **CURSOR FOR**

  SELECT propertyNo, street, city

  FROM PropertyForRent

  WHERE staffNo = ‘SL41’;

- EXEC SQL **OPEN** propertyCursor ;

- EXEC SQL **FETCH** propertyCursor
  INTO :propertyNo, :street, :city;

- EXEC SQL **CLOSE** propertyCursor ;

Using Cursors to Modify Data

- **DECLARE** cursor statement:

EXEC SQL DECLARE cursorName CURSOR FOR selectStatement

FOR UPDATE OF columnName[, …]

- The cursor-based **UPDATE** statement:

EXEC SQL UPDATE TableName

SET columnName = dataValue [, …]

WHERE CURRENT OF cursorName

- The cursor-based **DELETE** statement:

EXEC SQL DELETE FROM TableName

WHERE CURRENT OF cursorName

Agenda

1. Programmatic SQL
2. Simple Embedded SQL Statements
3. SQL Communication Area
4. Host Language Variables
5. Cursors
6. Dynamic SQL

Static SQL

- **Static SQL** provides significant functionality for the application developer by allowing access to the database using the normal interactive SQL statement, with minor modification in some cases.

- In static SQL, the pattern of database access is fixed and can be ‘hard-code’ into the program.

Dynamic SQL

- In **dynamic SQL**, the pattern of database access is not fixed and is known only at runtime.
- The basic difference between static SQL and dynamic SQL is that Static SQL does not **allow host variables to be used in place of table names or column names**.
- For example, in static SQL we **cannot write**:

```
EXEC SQL BEGIN DECLARE SECTION
        char TableName[20];
EXEC SQL END DECLARE SECTION;
EXEC SQL INSERT INTO :TableName
        VALUES(……);
```

EXECUTE IMMEDIATE Statement

- The basic idea of dynamic SQL is to place the complete SQL statement to be executed in a host variable. The host variable is passed to the DBMS to be executed.
- The format of **EXECUTE IMMEDIATE** statement:

EXEC SQL EXECUTE IMMEDIATE **[*hostVariable* | *stringLiteral*]**

- The function of **EXECUTE IMMEDIATE** statement:

  This command allows the SQL statement stored in ***hostVariable***, or in the literal, ***stringLiterial***, to be executed.

Example of EXECUTE IMMEDIATE

- The static SQL:

```
EXEC SQL BEGIN DECLARE SECTION;
     float increment;
EXEC SQL END DECLARE SECTION;
EXEC SQL UPDATE staff SET salary = salary + : increment
                WHERE staffNo = 'SL21';
```

- replace it with the following dynamic SQL statement:

```
EXEC SQL BEGIN DECLARE SECTION;
     char buffer[100];
EXEC SQL END DECLARE SECTION;
Sprintf(buffer, "UPDATE staff SET salary = salary + %f
                    WHERE staffNo = 'SL21' ", increment);
EXEC SQL EXECUTE IMMEDIATE :buffer;
```

PREPARE and EXECUTE Statement

- Dynamic SQL provides an alternative approach for SQL statements that may be executed more than once, involving the use of two complementary statements: PREPARE and EXECUTE.
- The format of the PREPARE statement is:

  EXEC SQL PREPARE statementName

  FROM [**hostVariable | stringLiteral**]

- The format of the EXECUTE statement is:

  EXEC SQL EXECUTE statementName

  USING hostVariable[indicatorVariable][, …]

Example of PREPARE and EXECUTE

```
EXEC SQL BEGIN DECLARE SECTION;
      char buffer[100];
      float  newSalary;
      char  staffNo[6];
EXEC SQL END DECLARE SECTION;
Sprintf(buffer, "UPDATE staff  SET salary = :sal WHERE staffNo = :sn");
EXEC SQL PREPARE stmt FROM  :buffer;
do {
        printf ("Enter staff number:");
        scanf("%s", staffNo);
        printf ("Enter salary:");
        scanf("%f", newSalary);
        EXEC SQL EXECUTE stmt USING :newSalary, :staffNo;
        printf("Enter another (Y/N)?");
        scanf("%c", more);
}
until (more != 'Y') ;
```

Section Objectives

In this section you will learn:

① How SQL statement can be embedded in high-level programming language.

② The differences between static and dynamic embedded SQL.

③ How to write programs that use static embedded SQL statements.

④ How to write programs that use dynamic embedded SQL statements.

Questions?

Assignments

- **Multiple-Choice Quiz 4**

Prerequisites for Next Section

- **Readings:**
  - **Required:** Connolly and Begg, sections 29.2.5 - 29.2.8 and 29.3 - 29.6 (in third edition, sections 28.3 - 28.7 except 28.7.1).
  - **Required:** Connolly and Begg, sections 29.3.2 and 29.8 (in third edition, sections 28.4.2 and 28.9).
  - **Required:** Connolly and Begg, section 29.7 (in third edition, section 28.8).
  - **Required:** Connolly and Begg, section 8.2.5 – 8.2.7