204222 - Fundamentals of Database Systems

Chapter 13

Introduction to SQL Programming Techniques

Adapted for 204222

by Areerat Trongratsameethong

Chapter 13 Outline

- Database Programming: Techniques and Issues

- Embedded SQL, Dynamic SQL, and SQLJ

- Database Programming with Function Calls: SQL/CLI and JDBC

- Database Stored Procedures and SQL/PSM

- Comparing the Three Approaches

Introduction to SQL Programming Techniques

- **Database applications**
  - Host language
    - Java, C/C++/C#, COBOL, or some other programming language
  - Data sublanguage
    - SQL
- SQL standards
  - Continually evolving
  - Each DBMS vendor may have some variations from standard

Database Programming: Techniques and Issues

- **Interactive interface**
  - SQL commands typed directly into a monitor
- Execute **file of commands**
  - *@<filename>*
- **Application programs** or **database applications**
  - Used as canned transactions by the end users access a database
  - May have **Web interface**

Approaches to Database Programming

- **Embedding** database commands in a general-purpose programming language
  - Database statements identified by a special prefix
  - **Precompiler** or **preprocessor** scans the source program code
    - Identify database statements and extract them for processing by the DBMS
  - Called **embedded SQL**

Approaches to Database Programming (cont'd.)

- Using a library of database functions
  - **Library of functions** available to the host programming language
  - **Application programming interface (API)**
- Designing a brand-new language
  - **Database programming language** designed from scratch
- First two approaches are more common

Impedance Mismatch

- Differences between database model and programming language model
- **Binding** for each host programming language
  - Specifies for each attribute type the compatible programming language types
- Cursor or iterator variable
  - Loop over the tuples in a query result

Typical Sequence of Interaction in Database Programming

- Open a connection to database server

- Interact with database by submitting queries, updates, and other database commands

- Terminate or close connection to database

Embedded SQL, Dynamic SQL, and SQLJ

- **Embedded SQL**
  - C language
- **SQLJ**
  - Java language
- Programming language called **host language**

Retrieving Single Tuples with Embedded SQL

- `EXEC SQL`
  - Prefix
  - **Preprocessor** separates embedded SQL statements from host language code
  - Terminated by a matching `END-EXEC`
    - Or by a semicolon (;)
- **Shared variables**
  - Used in both the C program and the embedded SQL statements
  - Prefixed by a colon (:) in SQL statement

Retrieving Single Tuples with Embedded SQL (cont'd.)

```
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 ;
```

Figure 13.1
C program variables used in the embedded SQL examples E1 and E2.

Retrieving Single Tuples with Embedded SQL (cont'd.)

- ## Connecting to the database

```
CONNECT TO <server name>AS <connection name>
AUTHORIZATION <user account name and password> ;
```

- ## Change connection

```
SET CONNECTION <connection name> ;
```

- ## Terminate connection

```
DISCONNECT <connection name> ;
```

Retrieving Single Tuples with Embedded SQL (cont'd.)

- **SQLCODE** and **SQLSTATE** communication variables
  - Used by DBMS to communicate exception or error conditions
- SQLCODE variable
  - 0 = statement executed successfully
  - 100 = no more data available in query result
  - < 0 = indicates some error has occurred

Retrieving Single Tuples with Embedded SQL (cont'd.)

- SQLSTATE
  - String of five characters
  - '00000' = no error or exception
  - Other values indicate various errors or exceptions
  - For example, '02000' indicates 'no more data' when using SQLSTATE

```
     //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)     }
```

Figure 13.2
Program segment E1,
a C program segment
with embedded SQL.

Retrieving Multiple Tuples with Embedded SQL Using Cursors

- Cursor
  - Points to a single tuple (row) from result of query
- **OPEN CURSOR** command
  - Fetches query result and sets cursor to a position before first row in result
  - Becomes current row for cursor
- **FETCH** commands
  - Moves cursor to next row in result of query

Figure 13.3

Program segment E2, a C program segment that uses
cursors with embedded SQL for update purposes.

```
    //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 ;
```

Retrieving Multiple Tuples with Embedded SQL Using Cursors (cont'd.)

- **`FOR UPDATE OF`**
  - List the names of any attributes that will be updated by the program

- Fetch orientation
  - Added using value: `NEXT`, `PRIOR`, `FIRST`, `LAST`, `ABSOLUTE` i, and `RELATIVE` i

```
DECLARE <cursor name> [ INSENSITIVE ] [ SCROLL ] CURSOR
[ WITH HOLD ] FOR <query specification>
[ ORDER BY <ordering specification> ]
[ FOR READ ONLY | FOR UPDATE [ OF <attribute list> ] ] ;
```

Specifying Queries at Runtime Using Dynamic SQL

- **Dynamic SQL**
  - Execute different SQL queries or updates dynamically at runtime

- Dynamic update

- Dynamic query

```
//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 ;
     . . .
```

Figure 13.4
Program segment E3, a C program segment that uses dynamic SQL for updating a table.

SQLJ: Embedding SQL Commands in Java

- Standard adopted by several vendors for embedding SQL in Java
- Import several class libraries
- **Default context**
- Uses **exceptions** for error handling
  - `SQLException` is used to return errors or exception conditions

SQLJ: Embedding SQL Commands in Java (cont'd.)

```
    //Program Segment J1:
 1) ssn = readEntry("Enter a Social Security Number: ") ;
 2) try {
 3)    #sql { select Fname, Minit, Lname, Address, Salary
 4)       into :fname, :minit, :lname, :address, :salary
 5)       from EMPLOYEE where Ssn = :ssn} ;
 6) } catch (SQLException se) {
 7)      System.out.println("Social Security Number does not exist: " + ssn) ;
 8)      Return ;
 9)    }
10) System.out.println(fname + " " + minit + " " + lname + " " + address
       + " " + salary)
```

Figure 13.7
Program segment J1, a Java program segment with SQLJ.

Retrieving Multiple Tuples in SQLJ Using Iterators

- **Iterator**
  - Object associated with a collection (set or multiset) of records in a query result

- **Named iterator**
  - Associated with a query result by listing attribute names and types in query result

- **Positional iterator**
  - Lists only attribute types in query result

Figure 13.8

Program segment J2A, a Java program segment that uses a named iterator to print employee information in a particular department.

```
    //Program Segment J2A:
0)  dname = readEntry("Enter the Department Name: ") ;
1)  try {
2)    #sql { select Dnumber into :dnumber
3)        from DEPARTMENT where Dname = :dname} ;
4)  } catch (SQLException se) {
5)    System.out.println("Department does not exist: " + dname) ;
6)    Return ;
7)    }
8)  System.out.printline("Employee information for Department: " + dname) ;
9)  #sql iterator Emp(String ssn, String fname, String minit, String lname,
        double salary) ;
10) Emp e = null ;
11) #sql e = { select ssn, fname, minit, lname, salary
12)    from EMPLOYEE where Dno = :dnumber} ;
13) while (e.next()) {
14)    System.out.printline(e.ssn + " " + e.fname + " " + e.minit + " " +
          e.lname + " " + e.salary) ;
15) } ;
16) e.close() ;
```

Retrieving Multiple Tuples in SQLJ Using Iterators (cont'd.)

Figure 13.9
Program segment J2B, a Java program segment that uses a positional iterator to print employee information in a particular department.

```
    //Program Segment J2B:
 0)  dname = readEntry("Enter the Department Name: ") ;
 1)  try {
 2)     #sql { select Dnumber into :dnumber
 3)        from DEPARTMENT where Dname = :dname} ;
 4)  } catch (SQLException se) {
 5)     System.out.println("Department does not exist: " + dname) ;
 6)     Return ;
 7)     }
 8)  System.out.printline("Employee information for Department: " + dname) ;
 9)  #sql iterator Emppos(String, String, String, String, double) ;
10)  Emppos e = null ;
11)  #sql e = { select ssn, fname, minit, lname, salary
12)     from EMPLOYEE where Dno = :dnumber} ;
13)  #sql { fetch :e into :ssn, :fn, :mi, :ln, :sal} ;
14)  while (!e.endFetch()) {
15)     System.out.printline(ssn + " " + fn + " " + mi + " " + ln + " " + sal) ;
16)     #sql { fetch :e into :ssn, :fn, :mi, :ln, :sal} ;
17)  } ;
18)  e.close() ;
```

Database Programming with Function Calls: SQL/CLI & JDBC

- Use of function calls
  - **Dynamic** approach for database programming
- Library of functions
  - Also known as **application programming interface (API)**
  - Used to access database
- **SQL Call Level Interface (SQL/CLI)**
  - Part of SQL standard

SQL/CLI: Using C as the Host Language

- **Environment record**
  - Track one or more database connections
  - Set environment information
- **Connection record**
  - Keeps track of information needed for a particular database connection
- **Statement record**
  - Keeps track of the information needed for one SQL statement

SQL/CLI: Using C as the Host Language (cont'd.)

- **Description record**
  - Keeps track of information about tuples or parameters

- **Handle** to the record
  - C pointer variable makes record accessible to program

Figure 13.11
Program segment CLI2, a C program segment that uses SQL/CLI
for a query with a collection of tuples in its result.

```
    //Program Segment CLI2:
 0) #include sqlcli.h ;
 1) void printDepartmentEmps() {
 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 Dno = ?",
        SQL_NTS) ;
11) prompt("Enter the Department Number: ", dno) ;
12) SQLBindParameter(stmt1, 1, SQL_INTEGER, &dno, 4, &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)    while (!ret2) {
19)       printf(lname, salary) ;
20)       ret2 = SQLFetch(stmt1) ;
21)       }
22)    }
23) }
```

JDBC: SQL Function Calls for Java Programming

- JDBC
  - Java function libraries
- Single Java program can connect to several different databases
  - Called data sources accessed by the Java program
- `Class.forName("oracle.jdbc.driver.OracleDriver")`
  - Load a **JDBC driver** explicitly

Figure 13.12

Program segment JDBC1, a Java program segment 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 password:") ;
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.printline(lname + salary) ;
24)     } }
25)  }
```

29

JDBC: SQL Function Calls for Java Programming

- **`Connection` object**
- **`Statement` object** has two subclasses:
  - `PreparedStatement` and `CallableStatement`
- Question mark (?) symbol
  - Represents a statement parameter
  - Determined at runtime
- **`ResultSet` object**
  - Holds results of query

Database Stored Procedures and SQL/PSM

- **Stored procedures**
  - Program modules stored by the DBMS at the database server
  - Can be functions or procedures
- SQL/PSM (**SQL/Persistent Stored Modules**)
  - Extensions to SQL
  - Include general-purpose programming constructs in SQL

Database Stored Procedures and Functions

- **Persistent stored modules**
  - Stored persistently by the DBMS

- Useful:
  - When database program is needed by several applications
  - To reduce data transfer and communication cost between client and server in certain situations
  - To enhance modeling power provided by views

Database Stored Procedures and Functions (cont'd.)

- Declaring stored procedures:

```
CREATE PROCEDURE <procedure name> (<parameters>)
<local declarations>
<procedure body> ;
declaring a function, a return type is necessary,
   so the declaration form is
CREATE FUNCTION <function name> (<parameters>)
RETURNS <return type>
<local declarations>
<function body> ;
```

Database Stored Procedures and Functions (cont'd.)

- Each parameter has parameter type
  - **Parameter type**: one of the SQL data types
  - **Parameter mode**: IN, OUT, or INOUT

- Calling a stored procedure:

  ```
  CALL <procedure or function name>
  (<argument list>) ;
  ```

SQL/PSM: Extending SQL for Specifying Persistent Stored Modules

- Conditional branching statement:

```
IF <condition> THEN <statement list>
ELSEIF <condition> THEN <statement list>
...
ELSEIF <condition> THEN <statement list>
ELSE <statement list>
END IF ;
```

SQL/PSM (cont'd.)

- Constructs for looping

```
WHILE <condition> DO
        <statement list>
END WHILE ;
REPEAT
        <statement list>
UNTIL <condition>
END REPEAT ;

FOR <loop name> AS <cursor name> CURSOR FOR <query> DO
        <statement list>
END FOR ;
```

SQL/PSM (cont'd.)

```
        //Function PSM1:
0)      CREATE FUNCTION Dept_size(IN deptno INTEGER)
1)      RETURNS VARCHAR [7]
2)      DECLARE No_of_emps INTEGER ;
3)      SELECT COUNT(*) INTO No_of_emps
4)      FROM EMPLOYEE WHERE Dno = deptno ;
5)      IF No_of_emps > 100 THEN RETURN "HUGE"
6)        ELSEIF No_of_emps > 25 THEN RETURN "LARGE"
7)        ELSEIF No_of_emps > 10 THEN RETURN "MEDIUM"
8)        ELSE RETURN "SMALL"
9)      END IF ;
```

Figure 13.14
Declaring a function in SQL/PSM.

Comparing the Three Approaches

- Embedded SQL Approach
  - Query text checked for syntax errors and validated against database schema at compile time
  - For complex applications where queries have to be generated at runtime
    - Function call approach more suitable

Comparing the Three Approaches (cont'd.)

- Library of Function Calls Approach
  - More flexibility
  - More complex programming
  - No checking of syntax done at compile time
- Database Programming Language Approach
  - Does not suffer from the impedance mismatch problem
  - Programmers must learn a new language

Summary

- Techniques for database programming
  - Embedded SQL
  - SQLJ
  - Function call libraries
  - SQL/CLI standard
  - JDBC class library
  - Stored procedures
  - SQL/PSM