Real SQL Programming

Embedded SQL
Call-Level Interface
Java Database Connectivity

SQL in Real Programs

◆ We have seen only how SQL is used at the generic query interface --- an environment where we sit at a terminal and ask queries of a database.
◆ Reality is almost always different.
  * Programs in a conventional language like C are written to access a database by "calls" to SQL statements.

Host Languages

◆ Any conventional language can be a host language, that is, a language in which SQL calls are embedded.
◆ The use of a host/SQL combination allows us to do anything computable, yet still get the very-high-level SQL interface to the database.

Connecting SQL to the Host Language

1. Embedded SQL is a standard for combining SQL with seven languages.
2. CLI (Call-Level Interface) is a different approach to connecting C to an SQL database.
3. JDBC (Java Database Connectivity) is a way to connect Java with an SQL database.

Embedded SQL

◆ Key idea: Use a preprocessor to turn SQL statements into procedure calls that fit with the host-language code surrounding.
◆ All embedded SQL statements begin with EXEC SQL, so the preprocessor can find them easily.

Shared Variables

◆ To connect SQL and the host-language program, the two parts must share some variables.
◆ Declarations of shared variables are bracketed by:

```
BEGIN DECLARE SECTION;
:host-language declarations>
END DECLARE SECTION;
```
Use of Shared Variables

- In SQL, the shared variables must be preceded by a colon.
  - They may be used as constants provided by the host-language program.
  - They may get values from SQL statements and pass those values to the host-language program.
- In the host language, shared variables behave like any other variable.

Example: Looking Up Prices

- We’ll use C with embedded SQL to sketch the important parts of a function that obtains a beer and a bar, and looks up the price of that beer at that bar.
- Assumes database has our usual Sells(bar, beer, price) relation.

Example: C Plus SQL

```sql
EXEC SQL BEGIN DECLARE SECTION;
char theBar;
float thePrice;
EXEC SQL END DECLARE SECTION;
/* obtain values for theBar and theBeer */
EXEC SQL SELECT INTO just like PSM
/* do something with thePrice */
```

Embedded Queries

- Embedded SQL has the same limitations as PSM regarding queries:
  - You may use SELECT-INTO for a query guaranteed to produce a single tuple.
  - Otherwise, you have to use a cursor.
  - Small syntactic differences between PSM and Embedded SQL cursors, but the key ideas are identical.

Cursor Statements

- Declare a cursor c with:
  EXEC SQL DECLARE c CURSOR FOR <query>;
- Open and close cursor c with:
  EXEC SQL OPEN CURSOR c;
  EXEC SQL CLOSE CURSOR c;
- Fetch from c by:
  EXEC SQL FETCH c INTO <variable(s)>;
  - Macro NOT FOUND is true if and only if the FETCH fails to find a tuple.

Example -- 1

- Let’s write C + SQL to print Joe’s menu — the list of beer-price pairs that we find in Sells(bar, beer, price) with bar = Joe’s Bar.
- A cursor will visit each Sells tuple that has bar = Joe’s Bar.
Example – 2 (Declarations)

EXEC SQL BEGIN DECLARE SECTION;
  char theBeer[21]; float thePrice;
EXEC SQL END DECLARE SECTION;

The cursor declaration goes outside the declare-section

Example – 3 (Executable)

EXEC SQL OPEN CURSOR c;
{
  EXEC SQL FETCH c INTO :theBeer, :thePrice;
/* format and print theBeer and thePrice */
}
EXEC SQL CLOSE CURSOR c;

Need for Dynamic SQL

- Most applications use specific queries and modification statements in their interaction with the database.
  - Thus, we can compile the EXEC SQL ... statements into specific procedure calls and produce an ordinary host-language program that uses a library.
- What if the program is something like a generic query interface, that doesn't know what it needs to do until it runs?

Dynamic SQL

- Preparing a query:
  EXEC SQL PREPARE <query-name>
    FROM <text of the query>;
- Executing a query:
  EXEC SQL EXECUTE <query-name>;
- "Prepare" = optimize query.
- Prepare once, execute many times.

Example: A Generic Interface

EXEC SQL BEGIN DECLARE SECTION;
  char query[MAX_LENGTH];
EXEC SQL END DECLARE SECTION;
while(1) {
  /* issue SQL> prompt */
  /* read user's query into array query */
  EXEC SQL PREPARE FROM :query;
  EXEC SQL EXECUTE
}

Execute-Immediate

- If we are only going to execute the query once, we can combine the PREPARE and EXECUTE steps into one.
- Use:
  EXEC SQL EXECUTE IMMEDIATE <text>;
Example: Generic Interface Again

EXEC SQL BEGIN DECLARE SECTION;
    char query[MAX_LENGTH];
EXEC SQL END DECLARE SECTION;
while(1) {
    /* issue SQL> prompt */
    /* read user's query into array query */
    EXEC SQL EXECUTE IMMEDIATE :query;
} 

SQL/CLI

♦ Instead of using a preprocessor, we can use a library of functions and call them as part of an ordinary C program.
    ♦ The library for C is called SQL/CLI = "Call-Level Interface."
    ♦ Embedded SQL's preprocessor will translate the EXEC SQL ... statements into CLI or similar calls, anyway.

Data Structures

♦ C connects to the database by structs of the following types:
    1. Environments: represent the DBMS installation.
    2. Connections: logins to the database.
    3. Statements: records that hold SQL statements to be passed to a connection.
    4. Descriptions: records about tuples from a query or parameters of a statement.

Environments, Connections, and Statements

♦ Function SQLAllocHandle(T,I,O) is used to create these structs, which are called environment, connection, and statement handles.
    ♦ T = type, e.g., SQL_HANDLE_STMT.
    ♦ I = input handle = struct at next higher level (statement < connection < environment).
    ♦ O = (address of) output handle.

Example: SQLAllocHandle

SQLAllocHandle(SQL_HANDLE_STMT, myCon, &myStat);
♦ myCon is a previously created connection handle.
♦ myStat is the name of the statement handle that will be created.

Preparing and Executing

♦ SQLPrepare(H, S, L) causes the string $S$, of length $L$, to be interpreted as an SQL statement, optimized, and the executable statement is placed in statement handle $H$.
♦ SQLExecute(H) causes the SQL statement represented by statement handle $H$ to be executed.
Example: Prepare and Execute

MySQL prepares the statement "SELECT beer, price FROM Sells WHERE bar = 'Joe's Bar'",
and executes it.

Dynamic Execution

- If we will execute a statement \( S \) only once, we can combine PREPARE and EXECUTE with:
  - `SQLExecuteDirect(H,S,L)`;
- As before, \( H \) is a statement handle and \( L \) is the length of string \( S \).

Fetching Tuples

- When the SQL statement executed is a query, we need to fetch the tuples of the result.
- That is, a cursor is implied by the fact we executed a query, and need not be declared.
- `SQLFetch(H)` gets the next tuple from the result of the statement with handle \( H \).

Accessing Query Results

- When we fetch a tuple, we need to put the components somewhere.
- Thus, each component is bound to a variable by the function SQLBindCol.
- This function has 6 arguments, of which we shall show only 1, 2, and 4:
  1. \( 1 \) = handle of the query statement.
  2. \( 2 \) = column number.
  3. \( 4 \) = address of the variable.

Example: Binding

- Suppose we have just done
  `SQLExecute(myStat), where myStat is the handle for query`
- `SELECT beer, price FROM Sells WHERE bar = 'Joe's Bar'
- Bind the result to theBeer and thePrice:
  `SQLBindCol(myStat, 1, &theBeer, , );`
  `SQLBindCol(myStat, 2, , &thePrice, );`

Example: Fetching

- Now, we can fetch all the tuples of the answer by:
  `while ( SQLFetch(myStat) != NULL )`;
  `{ /* do something with theBeer and thePrice */ }`
JDBC

- Java Database Connectivity (JDBC) is a library similar to SQL/CLI, but with Java as the host language.
- JDBC/CLI differences are often related to the object-oriented style of Java, but there are other differences.

Environments, Connections, and Statements

- The same progression from environments to connections to statements that we saw in CLI appears in JDBC.
- A connection object is obtained from the environment in a somewhat implementation-dependent way.
- We’ll start by assuming we have myCon, a connection object.

Statements

- JDBC provides two classes:
  1. Statement = an object that can accept a string that is an SQL statement and can execute such a string.
  2. PreparedStatement = an object that has an associated SQL statement ready to execute.

Creating Statements

- The Connection class has methods to create Statements and PreparedStatement.

```java
Statement stmt1 = myCon.createStatement();
PreparedStatement stmt2 = myCon.prepareStatement("SELECT beer, price FROM Sells" +
        "WHERE bar = 'Joe's Bar' " +
    );
createStatement with no argument returns a Statement with one argument it returns a PreparedStatement.
```

Executing SQL Statements

- JDBC distinguishes queries from modifications, which it calls “updates.”
- Statement and PreparedStatement each have methods executeQuery and executeUpdate.
  - For Statements, these methods have one argument: the query or modification to be executed.
  - For PreparedStatements: no argument.

Example: Update

- `stmt1` is a Statement.
- We can use it to insert a tuple as:

  ```java
  stmt1.executeUpdate("INSERT INTO Sells" +
        "VALUES('Brass Rail', 'Bud', 3.00)"
    );
  ```
Example: Query

- stat2 is a PreparedStatement holding the query "SELECT beer, price FROM Sells WHERE bar = 'Joe's Bar'.
- executeQuery returns an object of class ResultSet. We'll examine it later.
- The query:
  ResultSet Menu = stat2.executeQuery();

Accessing the ResultSet

- An object of type ResultSet is something like a cursor.
- Method next() advances the "cursor" to the next tuple.
  * The first time next() is applied, it gets the first tuple.
  * If there are no more tuples, next() returns the value FALSE.

Accessing Components of Tuples

- When a ResultSet is referring to a tuple, we can get the components of that tuple by applying certain methods to the ResultSet.
- Method getX(i), where X is some type, and i is the component number, returns the value of that component.
  * The value must have type X.

Example: Accessing Components

- Menu is the ResultSet for the query "SELECT beer, price FROM Sells WHERE bar = 'Joe's Bar'.
- Access the beer and price from each tuple by:
  while ( Menu.next() ) {
    theBeer = Menu.getString(1);
    thePrice = Menu.getFloat(2);
    /* do something with theBeer and thePrice */
  }