Database Management Systems - I, CS 157A

Embedded SQL, Dynamic SQL and CLI

Agenda

- Embedded SQL
  - Shared Variables
  - Cursor Statements
  - Declarations
  - Execution
- Dynamic SQL
- Call-Level Interface (CLI)
- Summary

Embedded SQL

Embedded SQL

- All embedded SQL statements begin with **EXEC SQL**, so the preprocessor can find them easily.
- **Key idea:** you need a pre-processor that converts “host-language + SQL” into “pure host-language;” it turns SQL statements into host-language-procedure calls (e.g., C procedure call) that fit with the surrounding host-language code.

Shared Variables

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

```
EXEC SQL  BEGIN  DECLARE SECTION;
      <host-language declarations>
EXEC SQL  END  DECLARE  SECTION ;
```

Always needed

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.
- Find the price for a given beer at a given bar?

Example: C Plus SQL

```
EXEC SQL BEGIN DECLARE SECTION;
   char theBar[21], theBeer[21];
   float thePrice;
EXEC SQL END DECLARE SECTION;
/* obtain values for theBar and theBeer */
EXEC SQL SELECT price INTO :thePrice
   FROM Sells
   WHERE bar = :theBar AND beer = :theBeer;
/* do something with thePrice */
```

Note 21-char arrays needed for 20 chars + End marker

SELECT-INTO as in PSM

Embedded Queries

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

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: Print Joe’s Menu

- 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: Declarations

```
EXEC SQL BEGIN  DECLARE  SECTION;
   char  theBeer[21];  float  thePrice;
EXEC SQL END  DECLARE  SECTION;
EXEC SQL DECLARE  c  CURSOR  FOR
   SELECT beer, price FROM Sells
   WHERE bar = ’Joe’’s Bar’;
```

The cursor declaration goes outside the declare-section

Example: Executable Part

```
EXEC SQL OPEN CURSOR  c;
while(1) {
   EXEC SQL  FETCH c
                  INTO :theBeer, :thePrice;
   if (NOT FOUND) break;
   /* format and print theBeer and thePrice */
}
EXEC SQL CLOSE CURSOR  c;
```

The C style of breaking loops

Dynamic SQL

Motivation for Dynamic SQL

- Most applications use specific queries and modification (IUD) statements to interact with the database.
  - The DBMS compiles **EXEC SQL** … statements into specific CLI procedure calls and produces an ordinary host-language program that uses a library.
- What about sqlplus, which 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) {                              // C-code
   /* issue SQL> prompt */
   /* read user's query into array query */
   EXEC SQL PREPARE q  FROM :query;
   EXEC SQL EXECUTE q;
}
```

q is an SQL variable representing the optimized form of whatever statement is typed into :query

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;
}
```

Call-Level Interface (CLI)

Call-Level Interface (CLI)

- Library call allow you to create a *statement handle* = struct/handle that points to the query plan ROOT of the SQL statement.

  SQLPrepare(**myHandle**, “SQL stmt”, ….);
- Use SQLExecute(**myHandle**) to execute the query plan corresponding to that SQL statement.

- **Example:**

```
SQLPrepare(handle1, “SELECT beer, price
                     FROM   Sells
                     WHERE  bar = ‘Joe’’s Bar’”, … );
SQLExecute(handle1);
```

Call-Level Interface (CLI)

- **Fetching Data**:
  To obtain the data from the query executed:
  - Bind variables to the component numbers of the returned query result
  - Fetch using the handle of the query statement
- **Example:**

```
SQLPrepare(handle1, …….. );
SQLBindCol( handl1, 1, SQL_CHAR, &&theBar, … );
SQLBindCol( handl1, 2, SQL_REAL, &&thePrice, … );
SQLEXECUTE(handle1);
…
While (SQLFetch( handle1 != SQL_NO_DATA) {
      …
}
…
```

Summary

Summary

- **Embedded SQL:**
  - **Shared Variables:** To connect SQL and the host-language
  - Cursor Statements
  - Declarations
  - Execution
- **Dynamic SQL:**
  - Query at runtime
- Call-Level Interface (CLI)

END