SQL in Programming Languages

Slides derived from those by Jeffrey D. Ullman

SQL and Programming Languages

- The user does not want to execute SQL statement
- She wants to interact with an application targeted to her domain
  - Limited set of choices
  - Simple execution of complex operations
  - Graphical interface:
    - Simple data input
    - Nice data output (presentation)

Applications

- They are written in traditional programming languages:
  - C, C++, Java, Fortran, C#, Visual Basic, Cobol
- Host languages

Approaches

- Embedded SQL
  - Older approach (since the 70s)
- Call Level Interface (CLI)
  - Most recent
  - SQL/CLI, ODBC, JDBC

Embedded SQL

- In the embedded SQL approach the programmer inserts SQL statements directly in the source code of the host programming language
- A precompiler is used to translate the code so that SQL statements are translated into function/procedure calls of the specific DBMS API
- From a file containing embedded SQL to a file in the same language containing function calls

Concrete Examples

- In DB2 you can develop embedded SQL applications in the following host programming languages: C, C++, COBOL, FORTRAN, and REXX
- The DB2 precompiler is invoked with PREP (PRECOMPILE)
- In Postgres the preprocessor for C is called ECPG

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;
```

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: 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 */
```

Embedded 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;
EXEC SQL DECLARE c CURSOR FOR
    SELECT beer, price FROM Sells
    WHERE bar = 'Joe''s Bar';
```

Example – 3 (Executable)

```
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;
```

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 q FROM :query;
    EXEC SQL EXECUTE q;
}
```

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;
}
```

DB2 Example

```
#include <stdio.h>
      #include <stdlib.h>
      #include <string.h>
      #include <sqlenv.h>
      #include <sqlutil.h>

     EXEC SQL BEGIN DECLARE SECTION;
       short id;
       char name[10];
       short dept;
       double salary;
       char hostVarStmtDyn[50];
     EXEC SQL END DECLARE SECTION;
```

DB2 Example

```
int main()
    {
      int rc = 0;
      EXEC SQL INCLUDE SQLCA;

      /* connect to the database */
      printf("\n Connecting to database...");
      EXEC SQL CONNECT TO "sample";
      if (SQLCODE  <0)
      {
         printf("\nConnect Error:  SQLCODE = %d\n”,SQLCODE);
         goto connect_reset;
      }
      else
      {
         printf("\n Connected to database.\n");
      }
```

The SQLCA structure is updated after the execution of each SQL statement.

SQLCODE is a field of SQLCA that contains the result of the last operation

DB2 Example

```
/* execute an SQL statement (a query) using static SQL; copy the single row
      of result values into host variables*/
   EXEC SQL SELECT id, name, dept, salary
         INTO :id, :name, :dept, :salary
         FROM staff WHERE id = 310;
   if (SQLCODE  <0)
   {
      printf("Select Error:  SQLCODE = %d\n”,SQLCODE);
   }
   else
   {
      /* print the host variable values to standard output */
      printf("\n Executing a static SQL query statement, searching for
         \n the id value equal to 310\n");
      printf("\n ID   Name        DEPT       Salary\n");
      printf(" %d\t%s\t%d\t%f\n”,id,name,dept,salary);
   }
```

DB2 Example

```
strcpy(hostVarStmtDyn, "UPDATE staff
                        SET salary = salary + 1000
                        WHERE dept = ?");
    /* execute an SQL statement (an operation) using a host variable
       and DYNAMIC SQL*/
    EXEC SQL PREPARE StmtDyn FROM :hostVarStmtDyn;
    if (SQLCODE  <0)
    {
       printf("Prepare Error:  SQLCODE = %d\n”,SQLCODE);
    }
    else
    {
       EXEC SQL EXECUTE StmtDyn USING :dept;
    }
    if (SQLCODE  <0)
    {
       printf("Execute Error:  SQLCODE = %d\n”,SQLCODE);
    }
```

DB2 Example

```
/* Read the updated row using STATIC SQL and CURSOR */
        EXEC SQL DECLARE posCur1 CURSOR FOR
           SELECT id, name, dept, salary
           FROM staff WHERE id = 310;
        if (SQLCODE  <0)
        {
           printf("Declare Error:  SQLCODE = %d\n”,SQLCODE);
        }
        EXEC SQL OPEN posCur1;
        EXEC SQL FETCH posCur1 INTO :id, :name, :dept, :salary ;
```

DB2 Example

```
if (SQLCODE  <0)
        {
           printf("Fetch Error:  SQLCODE = %d\n”,SQLCODE);
        }
        else
        {
           printf(" Executing an dynamic SQL statement, updating the
                    \n salary value for the id equal to 310\n");
           printf("\n ID   Name        DEPT       Salary\n");
            printf(" %d\t%s\t%d\t%f\n”,id,name,dept.salary);
        }
        EXEC SQL CLOSE posCur1;
```

DB2 Example

```
/* Commit the transaction */
        printf("\n  Commit the transaction.\n");
        EXEC SQL COMMIT;
        if (SQLCODE  <0)
        {
           printf("Error:  SQLCODE = %d\n”,SQLCODE);
        }
        /* Disconnect from the database */
        connect_reset :
           EXEC SQL CONNECT RESET;
           if (SQLCODE  <0)
           {
              printf("Connection Error:  SQLCODE = %d\n”,SQLCODE);
           }
        return 0;
       } /* end main */
```

Call Level Interface

- Sending commands to DBMS by means of function calls of an API
  - standard SQL/CLI (’95 and then part of SQL:1999)
  - ODBC: proprietary (Microsoft) implementation of SQL/CLI
  - OLE DB: high level API
  - ADO: higher level API
  - JDBC: CLI for Java

SQL/CLI

- SQL/CLI is the library for C
- ODBC differs from SQL/CLI in minor details

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);

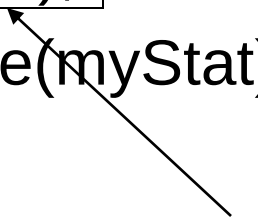
- myCon is a previously created connection handle.
- myStat is the name of the statement handle that will be created.

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

```
SQLPrepare(myStat, "SELECT beer, price
    FROM Sells WHERE bar = 'Joe''s Bar'",
    SQL_NTS);
SQLExecute(myStat);
```



This constant says the second argument is a "null-terminated string"; i.e., figure out the length by counting characters.

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 = handle of the query statement.
    - 2 = column number.
    - 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:

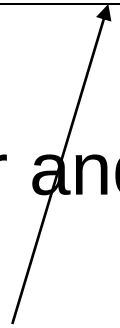
SQLBindCol(myStat, 1, , &theBeer, ,);

SQLBindCol(myStat, 2, , &thePrice, ,);

Example: Fetching

- Now, we can fetch all the tuples of the answer by:

```
while ( SQLFetch(myStat) != SQL_NO_DATA)
   {
        /* do something with theBeer and
            thePrice */
   }
```



CLI macro representing SQLSTATE = 02000 = “failed to find a tuple.”

OLE DB

- ODBC is complicated, so Microsoft proposed OLE DB and ADO
- OLE DB: is a library that provides applications with uniform access to data stored in diverse information sources
  - Not only relational
- OLE DB is based on the Microsoft object model: Component Object Model (COM)

ADO and ADO.NET

- ADO: Activex Data Object
- High level interface for OLE DB
- ADO.NET: ADO for the .NET framework
- ADO.NET is independent from OLE DB: there does not exist OLE DB.NET

.NET Framework

- The .NET Framework is Microsoft's replacement for COM technology.
- You can code .NET applications in over forty different programming languages. The most popular languages for .NET development are C# and Visual Basic .NET.
- The .NET Framework class library provides the building blocks with which you build .NET applications. This class library is language agnostic and provides interfaces to operating system and application services.

.NET Framework

- .NET applications (regardless of language) compile into Intermediate Language (IL), a type of bytecode.
- The Common Language Runtime (CLR) is the heart of the .NET Framework, compiling the IL code on the fly, and then running it.
- In running the compiled IL code, the CLR activates objects, verifies their security clearance, allocates their memory, executes them, and cleans up their memory once execution is finished.