**SQL APIs**

- Interactive SQL command interpreters (e.g., DB2’s command line processor) are simply domain-independent client programs that interact with an SQL database server.
- In general, it is necessary to write other client programs for specific applications.
- SQL has “bindings” for various programming languages (e.g., C/C++, Java) that describe how applications written in those languages can be made to interact with a database server.

The main problem is the “impedance mismatch” between set-oriented SQL and the application programming language. How should data be passed back forth between the two?

---

**Development Process for Embedded SQL Applications**

![Diagram of the development process for Embedded SQL applications]
A Simple Example

```c
#include <stdio.h>
EXEC SQL INCLUDE SQLCA;

main() {
    EXEC SQL WHENEVER SQLERROR GOTO error;
    EXEC SQL CONNECT TO sample;
    EXEC SQL UPDATE Employee
        SET salary = 1.1*salary
        WHERE empno = '000370';
    EXEC SQL COMMIT WORK;
    EXEC SQL CONNECT RESET;
    return(0);

error:
    printf("update failed, sqlcode = %ld\n",SQLCODE );
    return(-1);
}
```

Static Embedded SQL

- SQL DML and DDL can be embedded in a C program by prefixing with “EXEC SQL” and sufficing with “;”.
- host variables are used to send and receive values from the database system
  - values can be sent by using host variables in place of constants.
  - values can be received by using host variables in an INTO clause.

The SELECT statement is different in embedded SQL.
Declaring Host Variables

EXEC SQL BEGIN DECLARE SECTION;
char deptno[4];
char deptname[30];
char mgrno[7];
char admrdept[4];
char location[17];
EXEC SQL END DECLARE SECTION;

/* program assigns values to variables */

EXEC SQL INSERT INTO
    Department(deptno,deptname,mgrno,admrdept,location)
VALUES
    (:deptno,:deptname,:mgrno,:admrdept,:location);

Domain and Type Correspondence

<table>
<thead>
<tr>
<th>Domain</th>
<th>C Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>long int v;</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>short int v;</td>
</tr>
<tr>
<td>REAL</td>
<td>float v;</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>double v;</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>char v[n+1];</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>char v[n+1]; or</td>
</tr>
<tr>
<td></td>
<td>struct tag { short int len; char v[n]; }</td>
</tr>
<tr>
<td>DATE</td>
<td>char v[11];</td>
</tr>
</tbody>
</table>

Each SQL domain (type) corresponds to a type in the host language. See, e.g., the DB2 Application Development Guide for complete list.
**Queries Using INTO**

Print the last name of a specified employee.

```c
int PrintEmployeeName( char employeenum[] )
{
    EXEC SQL BEGIN DECLARE SECTION;
    char empno[7];
    char lastname[16];
    EXEC SQL END DECLARE SECTION;
    strcpy(empno,employeenum);
    EXEC SQL
        SELECT lastname INTO :lastname
        FROM employee
        WHERE empno = :empno;
    if( SQLCODE < 0 ) { return( -1 ); } /* error */
    else if( SQLCODE == 100) { printf("no such employee\n");}
    else { printf("%s\n",lastname); }
    return( 0 );
}
```

---

**Indicator Variables**

- What if a returned value is NULL?
  - NULLs are handled using special flags called *indicator variables*.
  - Any host variable that might receive a NULL should have a corresponding indicator variable.
  - In C/C++, indicator variables are *short ints*
Indicator Variables: An Example

```c
int PrintEmployeePhone( char employeenum[] ) {
    EXEC SQL BEGIN DECLARE SECTION;
    char empno[7];
    char phonenum[5];
    short int phoneind;
    EXEC SQL END DECLARE SECTION;
    strcpy(empno,employeenum);
    EXEC SQL
        SELECT phoneno INTO :phonenum :phoneind
        FROM employee WHERE empno = :empno;
    if( SQLCODE < 0 ) { return( -1 ); } /* error */
    else if( SQLCODE==100){printf("no such employee\n");}
    else if (phoneind<0){printf("phone unknown\n");}
    else { printf("%s\n",phonenum); }
    return( 0 );
}
```

Cursors

- If a query may return more than one row, then a *cursor* must be used to retrieve values from the result.

- A cursor is a bit like a pointer that refers to some row of the result. At any time, a cursor may be in one of three places:
  - before first tuple
  - on a tuple
  - after last tuple

```
<table>
<thead>
<tr>
<th></th>
<th>BEFORE FIRST TUPLE</th>
<th>-n - 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>-n</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>-n + 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n-1</td>
<td></td>
<td>-2</td>
</tr>
<tr>
<td>n</td>
<td></td>
<td>-1</td>
</tr>
<tr>
<td>n+1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Using Cursors

1. Declare the cursor
   - Declaring a cursor associates a cursor identifier with a query.

2. Open the cursor
   - Opening a cursor (conceptually) causes the query to be evaluated, generating a result.

3. Fetch one or more tuples using the cursor
   - Each call to the FETCH command returns values from one tuple of the generated result.

4. Close the cursor

The FETCH Command Syntax

FETCH [<location>] <cursor-name>
    [ INTO <host-var1>, <host-var2> ... ]

- Possible locations:
  - NEXT (this is the default)
  - PRIOR
  - FIRST
  - LAST
  - ABSOLUTE \( n \)
  - RELATIVE \( n \)
Using Cursors: An Example

```c
void PrintEmpNames() {
    int rval; /* -1 for error, 0 for success */
    EXEC SQL BEGIN DECLARE SECTION;
    char fullname[30];
    EXEC SQL END DECLARE SECTION;
    EXEC SQL DECLARE C1 CURSOR FOR
        SELECT firstnme || ' ' || lastname FROM Employee;
    EXEC SQL OPEN C1;
    for( ;; ) {
        EXEC SQL FETCH NEXT C1 INTO :fullname;
        if (SQLCODE == 100) { rval = 0; break; }
        else if (SQLCODE < 0) { rval = -1; break; }
        printf("%s\n", fullname );
    }
    EXEC SQL CLOSE C1;
    return(rval); }
```

Dynamic Embedded SQL

- Must be used when tables, columns or predicates are not known at the time the application is written.

- Basic idea:
  1. prepare the statement for execution: PREPARE
     - in static embedded SQL programs, statement preparation is handled at compile time by the preprocessor
  2. execute the prepared statement: EXECUTE

- once prepared, a statement may be executed multiple times, if desired
Dynamic Embedded SQL: A Simple Example

EXEC SQL BEGIN DECLARE SECTION;
char s[100] =
"INSERT INTO department VALUES ('000456','Legal',..)";
EXEC SQL END DECLARE SECTION;
EXEC SQL EXECUTE IMMEDIATE :s;

or, to factor cost of “preparing”

EXEC SQL BEGIN DECLARE SECTION;
char s[100] =
"INSERT INTO department VALUES ('000456','Legal',..)";
EXEC SQL END DECLARE SECTION;
EXEC SQL PREPARE S1 FROM :s;
EXEC SQL EXECUTE S1;
EXEC SQL EXECUTE S1;

Dynamic Embedded SQL: Using Host Variables for Input

EXEC SQL BEGIN DECLARE SECTION;
char s[100] = "INSERT INTO employee VALUES (?, ?, ... )";
char empno[7];
char firstname[13];
...
EXEC SQL END DECLARE SECTION;

EXEC SQL PREPARE S1 FROM :s;
strcpy(empno,"000111");
strcpy(firstname,"Ken");
...
EXEC SQL EXECUTE S1 USING :empno, :firstname, ... ;
Placeholders

- In the query string
  ```sql
  INSERT INTO employee VALUES (?, ?, ... )
  ```
  the ? are called placeholders
- placeholders can appear where literals can appear - not in place of relation names, column names, etc.
- host variable values replace the placeholders when the prepared statement is executed
- the USING clause is used to specify which host variables should replace the placeholders:
  ```sql
  EXEC SQL EXECUTE S1 USING :empno, :firstname, ...
  ```
- USING can only use used with previously-prepared statements, not with EXECUTE IMMEDIATE

Dynamic Single-Row Queries

```sql
EXEC SQL BEGIN DECLARE SECTION;
char s[100] =
  "select lastname,salary from employee where empno = ?"
char empno[7];
char lastname[16];
double salary;
short int salaryind;
EXEC SQL END DECLARE SECTION;
EXEC SQL PREPARE S1 FROM :s;
EXEC SQL EXECUTE S1
  INTO :lastname, :salary:salaryind USING :empno
```

- the INTO clause specifies which host variables receive the results
- INTO (with EXECUTE) in dynamic SQL is like INTO (with SELECT) in static
Dynamic Cursors

EXEC SQL BEGIN DECLARE SECTION;
char s[100] =
"select lastname,salary from employee where edlevel = ?"
short int edlevel;
char lastname[16];
double salary;
short int salaryind;
EXEC SQL END DECLARE SECTION;
EXEC SQL PREPARE S1 FROM :s;
EXEC SQL DECLARE C1 CURSOR FOR S1;
edlevel = 18;
EXEC SQL OPEN C1 USING :edlevel;
while( ... ) {
    EXEC SQL FETCH FROM C1
    INTO :lastname, :salary:salaryind;
}

Descriptors and the SQLDA

- if the numbers and types of input and output values are not known in advance, SQL descriptors can be used determine them at run-time
- an SQLDA (descriptor area) is used to hold a description of the structure (number of attributes and their types) of a query result.
- the DESCRIBE command can be used to populate a descriptor area, that is, no find out the structure of a query result
JDBC, ODBC and CLI

- CLI (Call-Level Interface) is a vendor-neutral ISO standard programming interface for SQL database systems. It is similar to ODBC.

- ODBC (Open Database Connectivity), popularized by Microsoft, is a programming interface for SQL database systems.

- JDBC (Java Database Connectivity) is a collection of Java classes that provide an ODBC/CLI-like programming interface.

- An embedded SQL program used to access one DBMS must be recompiled before it can be used to access a different DBMS.

- A CLI/ODBC/JDBC program need not be recompiled - a single application may even access multiple DBMS at the same time.

---

A CLI Example

```c
SQLHANDLE henv; /* an environment handle*/
SQLHANDLE hdbc; /* a connection handle */
SQLHANDLE hstmt; /* a statement handle */
SQLCHAR numteamsquery[] = "select count(*) from teams";
SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
DBconnect(henv, &hdbc, server, uid, pwd);
SQLAllocHandle( SQL_HANDLE_STMT, hdbc, &hstmt );
SQLExecDirect(hstmt, numteamsquery, SQL_NTS ); /* execute */
SQLFetch(hstmt); /* get one row of the result */
SQLGetData(hstmt, 1, SQL_C_LONG, &numteams, sizeof(numteams), &bytesremaining);
SQLFreeStmt(hstmt, SQL_CLOSE); /* close the statement */
```

---

CLI/ODBC interface is similar to dynamic embedded SQL