Embedded SQL

Davood Rafiei

Introduction

- Basic Idea: Use SQL statements inside a host language (C, C++, Java, Python, …).
- Advantages:
  - Can do all the fancy things you do in C/Java/Python.
  - Still have the power of SQL.
- Need mechanisms for
  - Embedding SQL statements in code
  - Transferring data to/from DBMS
  - Compiling and linking the program.

SQL inside Applications

- Statement-level interface
  - SQL statements appear as new statements in the program.
  - Precompile step: replaces new statements with some procedure calls.
  - Flavors: Static SQL, Dynamic SQL
- Call-level interface
  - Program is written entirely in the host language
  - No precompile step.
  - SQL statements are passed as parameters to some procedures.
  - Flavors: ODBC, JDBC, sqlite callback

Statement-level Interface

- New statements:
  - EXEC SQL <sql statement>
  - E.g.
    - ✓EXEC SQL SELECT …
    - ✓EXEC SQL UPDATE ...

- Static SQL: e.g.

```
EXEC SQL SELECT COUNT(*) INTO :cnt
                FROM emp;
```

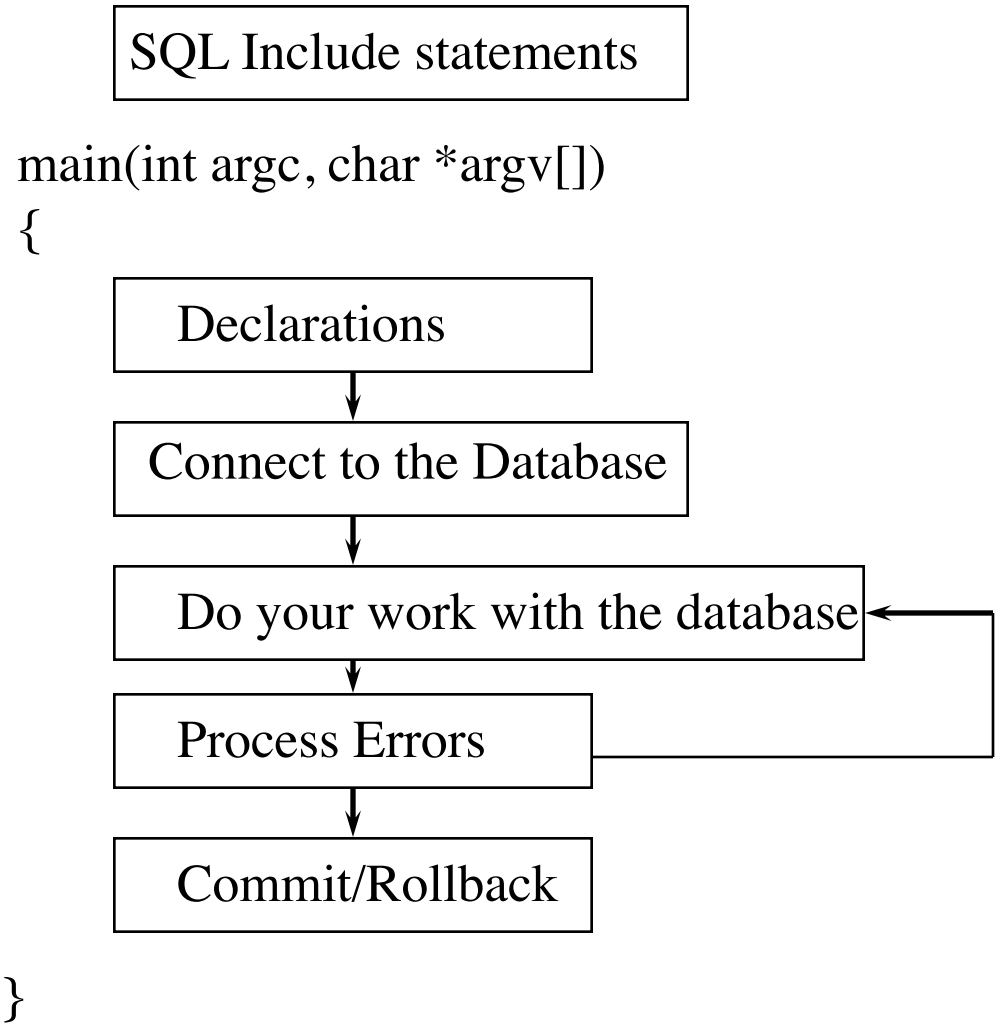
The SQL statement is known at compile time.

- Dynamic SQL: e.g.

```
strcpy(qstr, "SELECT COUNT(*) FROM emp");
EXEC SQL PREPARE Q FROM :qstr;
```

The SQL statement is not known at compile time.

Program Structure



SQL Include Statements

- An essential one: SQL Communication Area
  - **#include <sqlca.h>** , or
  - **EXEC SQL INCLUDE SQLCA**

- It provides runtime status information including:
  - sqlca.sqlcode: the return code of SQL statements
  - sqlca.sqlerrm: the error messages (if any)
    - ✓ sqlca.sqlerrm.sqlerrmc : error message
    - ✓ sqlca.sqlerrm.sqlerrml : the length of the error message

Host Variables

- Used to pass values between a SQL statement and the rest of the program.

- Declaration: as follows

```
EXEC SQL BEGIN DECLARE SECTION;
    /* all host variable declarations go here */
    int     cnt;
    char name[20];
EXEC SQL END DECLARE SECTION;
```

- Usage in SQL statements:

```
EXEC SQL SELECT COUNT(*) INTO :cnt FROM emp;
```

  - The variable must be preceded with `:' to distinguish it from SQL identifiers.

Error Handling

- Check sqlca.sqlcode, the return code of SQL statements
  - == 0 : the command was successful
  - > 0 : no row in the output
  - < 0 : error
- Use WHENEVER statements: e.g.

```
EXEC SQL WHENEVER SQLERROR      GOTO lbl1;
EXEC SQL WHENEVER NOT FOUND     GOTO lbl3;

EXEC SQL WHENEVER SQLERROR      DO err_func();

EXEC SQL WHENEVER SQLERROR      CONTINUE;
```

Error Handling

- Example:

```
EXEC SQL DROP TABLE emp;
if (sqlca.sqlcode == 0) {
  printf("Table emp is successfully dropped\n");
}
else {
  sqlca.sqlerrm.sqlerrmc[sqlca.sqlerrm.sqlerrml] = '\0';
  printf("Oracle error: %s\n", sqlca. sqlerrm.sqlerrmc);
}
```

myprog.pc: Our First Program

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

int main(int argc, char *argv[])
{
   EXEC SQL BEGIN DECLARE SECTION;
        char user[10] = "SCOTT";
        char pwd[10] = "TIGER";
   EXEC SQL END DECLARE SECTION;

   EXEC SQL WHENEVER SQLERROR  GOTO error;
   EXEC SQL CONNECT  :user  IDENTIFIED BY  :pwd;
   printf("connected to Oracle");

   /* SQL statements */

   EXEC SQL COMMIT RELEASE;
   return 0;
```

myprog.pc (Cont.)

```
error:
    sqlca.sqlerrm.sqlerrmc[sqlca.sqlerrm.sqlerrml] = '\0';
    printf("Oracle Error: %s\n", sqlca.sqlerrm.sqlerrmc);

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK RELEASE;
    return 1;
}
```

Program Preparation

- Precompile

  proc myprog.pc

  - result: myprog.c (a pure C program with SQL statements replaced with library calls)
- Compile the C program

  cc myprog.c

  - result: myprog.o
- Link the libraries

  cc -o myprog myprog.o -L…

  - result: myprog (an executable program)
- ** Use a makefile instead **
  - Check the Oracle directory for a sample *makefile*

myprog2.pc: Our 2nd Program

```
/* get the emp_number from input and print the emp_name */
…
EXEC SQL BEGIN DECLARE SECTION;
   int emp_number;
   char emp_name[30];
EXEC SQL END DECLARE SECTION;

EXEC SQL WHENEVER SQLERROR     GOTO  error;
EXEC SQL WHENEVER NOT FOUND    GOTO  nope;
/* Connect to the database ….. */

printf("Enter emp_number:");
scanf("%d", emp_number);

EXEC SQL SELECT ename INTO :emp_name
                FROM    emp
                WHERE empno = :emp_number;
printf("Employee name is %s\n", emp_name);
return 0;

error: …
nope: ...
```

Complications

- What if a host variable is assigned a NULL?
  - not a constant.
  - solution: use an extra indicator variable.
- What if a SQL statement returns more than one row?
  - cannot fit it in any C variable!
  - solution: use a cursor.

Indicator Variables

```
/* get the emp_number from input and print the emp_name */
…

EXEC SQL BEGIN DECLARE SECTION;
   int emp_number;
   char emp_name[30];
   short emp_name_ind;
EXEC SQL END DECLARE SECTION;

...

EXEC SQL SELECT ename INTO :emp_name  INDICATOR :emp_name_ind
                FROM    emp
                WHERE empno = :emp_number;
if (emp_name_ind <  0)
  printf("Employee name is NULL\n");
else
  printf("Employee name is %s\n", emp_name);

…
```

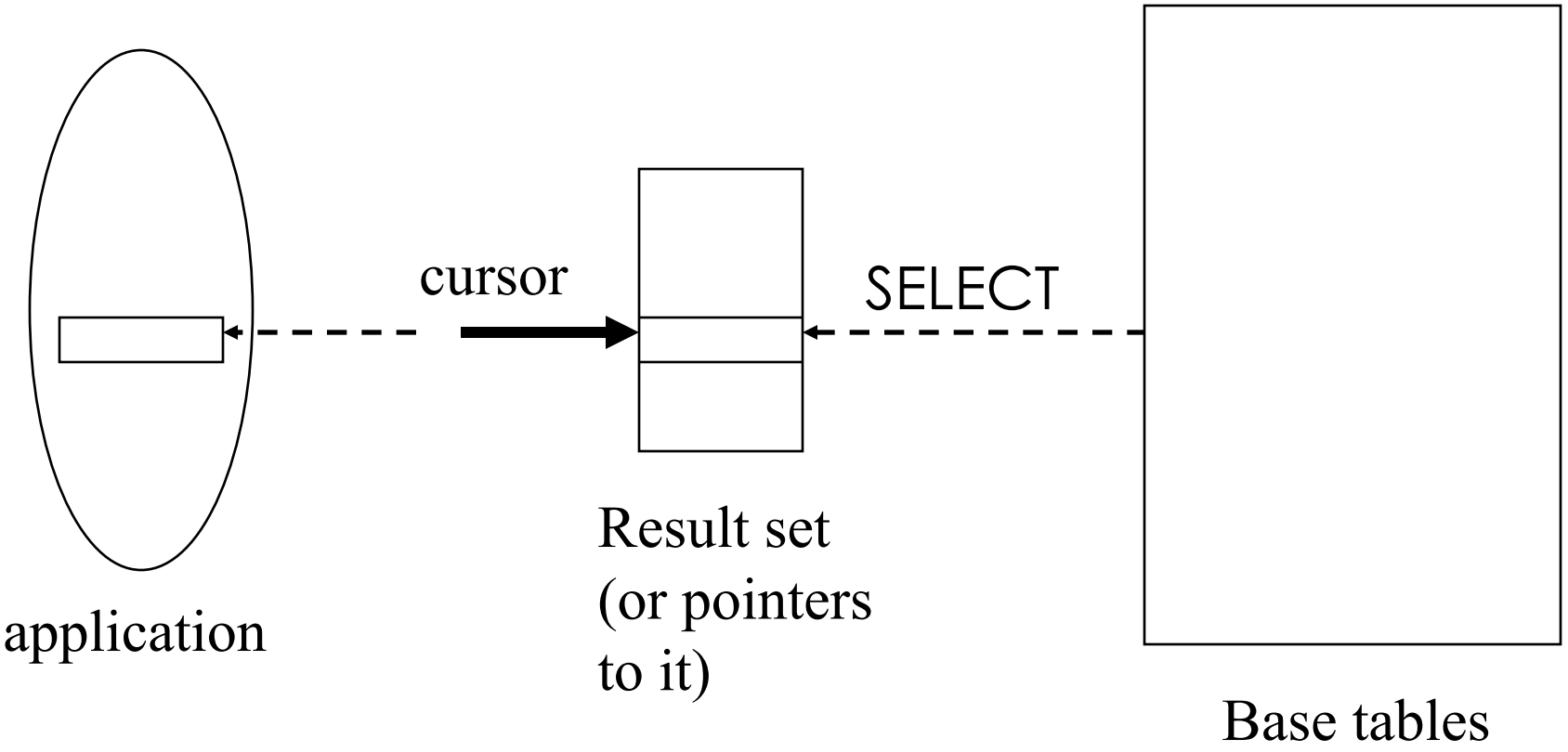
Use of a Cursor

```
/* print the names of all employees */
…
EXEC SQL BEGIN DECLARE SECTION;
   char emp_name[30];
EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE emp_cursor CURSOR FOR
                SELECT ename
                FROM    emp;

EXEC SQL OPEN emp_cursor;
EXEC SQL WHENEVER NOT FOUND  GOTO end;

printf("Employee names are:\n");
for (;;) {
   EXEC SQL FETCH emp_cursor INTO :emp_name;
    printf("%s\n", emp_name);
}
end:
   EXEC SQL CLOSE emp_cursor;
   EXEC SQL COMMIT RELEASE;
   return 0;
```

Cursor Overview

- ***Result set*** – set of rows produced by a SELECT statement
- ***Cursor*** – pointer to a row in the result set.
- Cursor operations:
  - *Declaration*
  - *Open* – execute SELECT to determine result set and initialize pointer
  - *Fetch* – advance pointer and retrieve next row
  - *Close* – deallocate cursor

Cursor


cursor
SELECT
application
Result set
(or pointers
to it)
Base tables

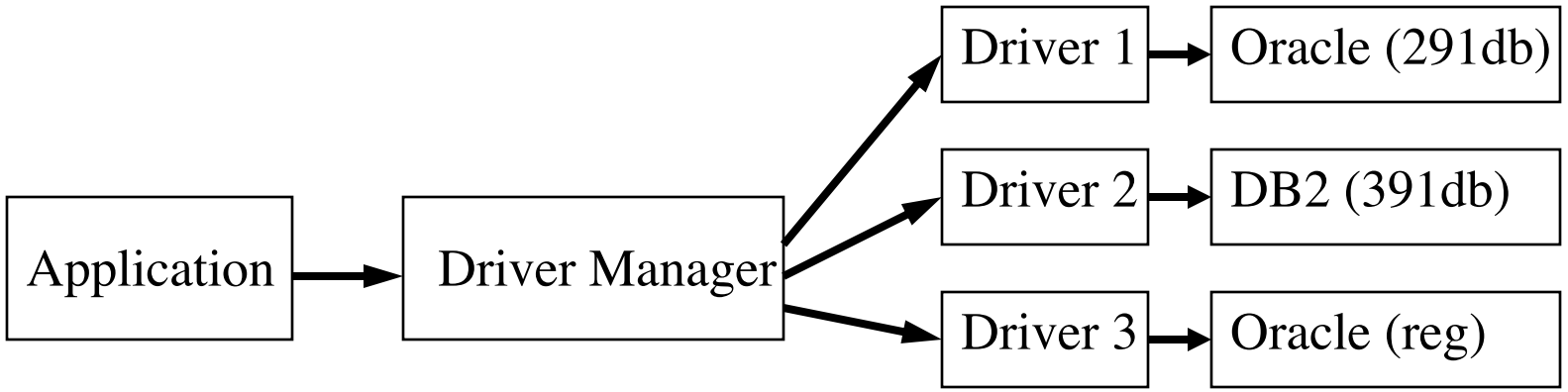
Statement-level Interface (Summary)

- One database at a time
- Both schema (for input and output) and database name must be known at compile time

- Ordered from the most efficient to the least
  - Static SQL
  - Dynamic SQL
  - Call-level interface: JDBC

Call-level Interface

- Neither the schema nor the database is known at compile time.
- The application can connect to more than one database.



Executing a Query

import java.sql.*; // *imports all classes in package* java.sql
String db_url = "jdbc:sqlite:/Users/drafiei/291dbfile.db";
String driverNamel = "org.sqlite.JDBC";
//String db_url = "jdbc:oracle:thin:@gwynne.cs.ualberta.ca:1521:CRS";
//String driverName = "oracle.jdbc.driver.OracleDriver";

Class.forName (*driverName*); // *loads the specified driver*

Connection con = DriverManager.getConnection(db_*url)*
//Connection con = DriverManager.getConnection(db_*url, Id, Pwd*);

- *connects to the DBMS at address db_url*
- *If successful, creates a connection object,* con, *for managing the connection*

Statement stat = con.CreateStatement ();

- *Creates a statement object* stat
- *Statements have* executeQuery() *method*

Executing a Query

String query = "SELECT T.*StudId* FROM Transcript T" +
"WHERE T.*CrsCode* = 'cse305' " +
"AND T.*Semester* = 'S2000' ";

ResultSet res = stat.executeQuery (query);

- *Creates a result set object,* res.
- *Prepares and executes the query.*
- *Stores the result set produced by execution in* res *(analogous to opening a cursor).*
- *The query string can be constructed at run time (as above).*
- *The input parameters are plugged into the query when the string is formed (as above)*

Result Sets and Cursors

- Three types of result sets in JDBC:
  - *Forward-only*: not scrollable
  - *Scroll-insensitive*: scrollable (can jump up and down in the result set!), changes made to underlying tables after the creation of the result set are not visible through that result set
  - *Scroll-sensitive*: scrollable, changes made to the tuples in a result set after the creation of that set are visible through the result set

Result Set

```
Statement stat = con.createStatement (
    ResultSet.TYPE_SCROLL_SENSITIVE,
    ResultSet.CONCUR_UPDATABLE  );
```

- Any result set type can be declared *read-only* or *updatable* – CONCUR_UPDATABLE (assuming SQL query satisfies the conditions for updatable views)
- Current row of an updatable result set can be updated or deleted, and a new row can be inserted, causing changes in base table

```
res.updateString ("Name", "John" );    // update attribute "Name" of
                                       // current row in row buffer.
or  res.updateRow ( );                 // make the change permanent
    res.CancelRowUpdate ( );           // cancel the update
```

SQLite callback in C

- Three functions: open, exec, close
  - sqlite3_open(const char *filename, sqlite3 **db)
  - sqlite3_exec(sqlite3 *db, const char *sql, sqlite_callback, void *data, char **errmsg)
  - sqlite3_close(sqlite3 *db)
- Output is processed by a callback function

myp.c

```
##include <stdio.h>
#include <stdlib.h>
#include <sqlite3.h>

static int callback(void *data, int argc, char **argv, char **aColName){
   int i;
   fprintf(stderr, "%s: ", (const char*)data);

   for(i = 0; i<argc; i++){
      printf("%s = %s\n", aColName[i], argv[i] ? argv[i] : "NULL");
   }

   printf("\n");
   return 0;
}
```

```
int main(int argc, char* argv[]) {
   sqlite3 *db;   char *zErrMsg = 0;
   int rc;  char *sql;
   const char* data = "Callback function called";

   /* Open database */
   rc = sqlite3_open("291lect.db", &db);
   if( rc ) {
      fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
      return(0);
   } else {
      fprintf(stderr, "Opened database successfully\n");
   }

   /* Create and execute SQL statement */
   sql = "SELECT * from customer";
   rc = sqlite3_exec(db, sql, callback, (void*)data, &zErrMsg);

   if( rc != SQLITE_OK ) {
      fprintf(stderr, "SQL error: %s\n", zErrMsg);
      sqlite3_free(zErrMsg);
   } else {
      fprintf(stdout, "Operation done successfully\n");
   }
   sqlite3_close(db);
   return 0;
}
```

Compile and Run

- Compile
  gcc -Wall -std=c99 -L/usr/lib/sqlite3 myp.c -lsqlite3 -o myp
- Run
  ./myp

Opened database successfully
Callback function called: cname = Davood
street = 114 St
city = Edmonton

Callback function called: cname = Ehsan
street = University Ave
city = NULL

Operation done successfully

More information
https://www.sqlite.org/c_interface.html

More Information

- Statement-level Interface:
  - Not supported in SQLite
  - But is supported in other databases such as Oracle
- Call-level Interface
  - JDBC tutorials
  - Callback interface in C

Summary

- Covered:
  - Static SQL
  - Call level interface (JDBC, C)
- SQLite in Python is covered in the lab
- Not Covered: Dynamic SQL, ODBC
- Final note (but quite important):
  - Avoid data processing in the host language if it can be passed to SQL.
  - Use the host language for things that cannot be done in SQL.