SQL Inside Applications

Davood Rafiei

Introduction

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

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 (and can be compiled or executed)
  - SQL statements are passed as parameters to some functions.
  - 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;
```

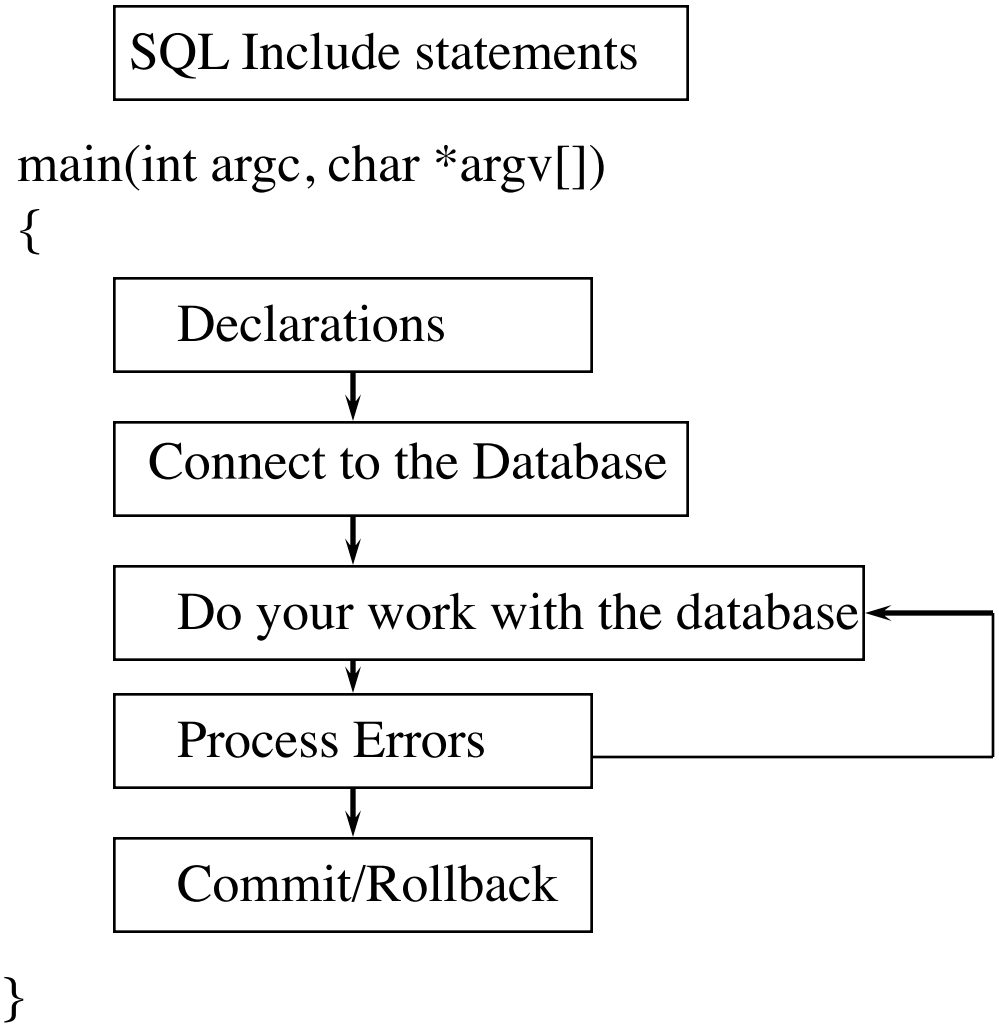
The SQL statement is known at compile time.

- 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



Host Variables

- Pass values between a SQL statement and the rest of the program.

- Declaration: as follows

```
EXEC SQL BEGIN DECLARE SECTION;
    int     cnt;
    char name[20];
EXEC SQL END DECLARE SECTION;
```

- Usage in SQL statements:

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

Colon distinguishes host variables 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

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: ...
```

Complication

- What if a SQL statement returns more than one row?
  - cannot fit it in any C variable!
  - solution: use a cursor.

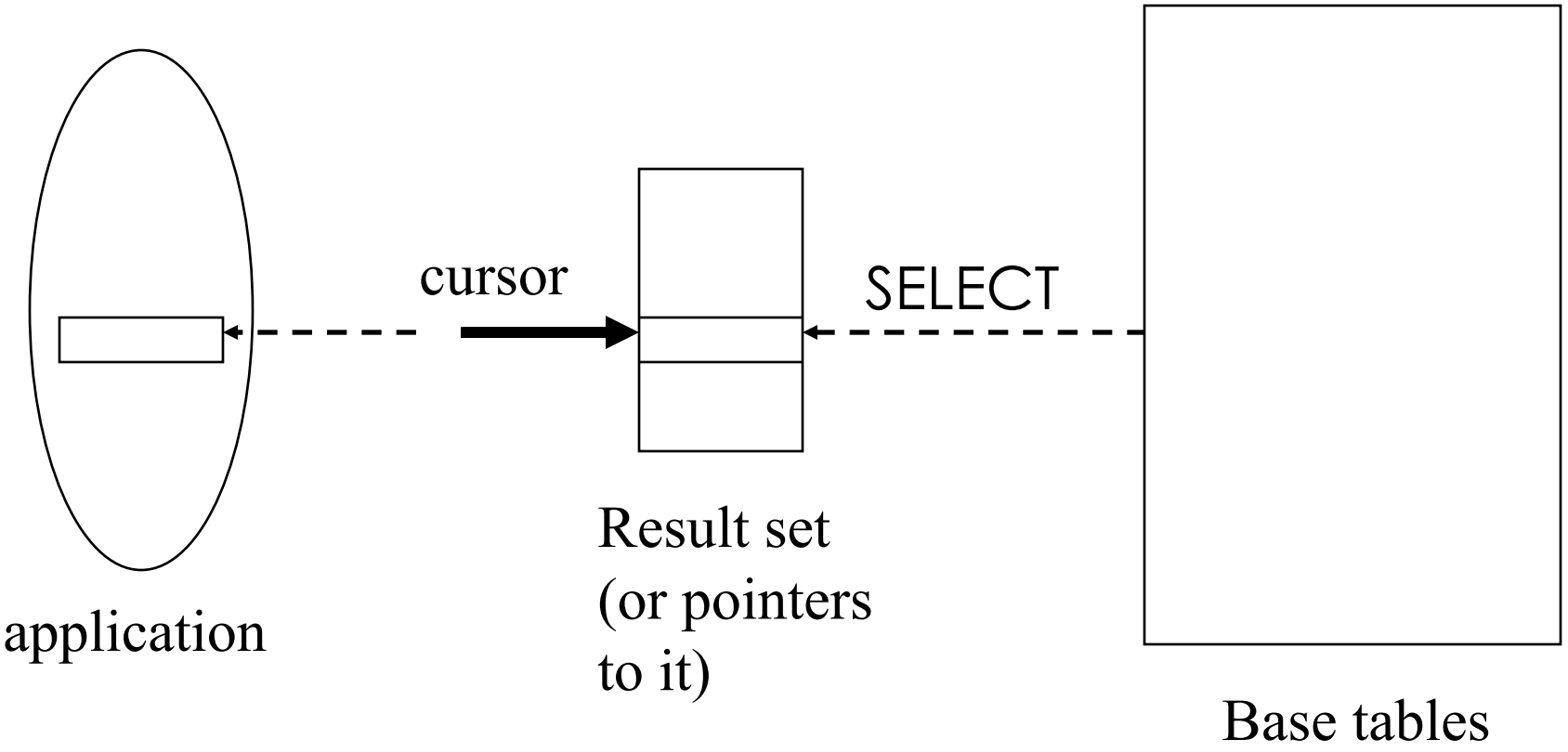
Use of a Cursor

```
/* 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

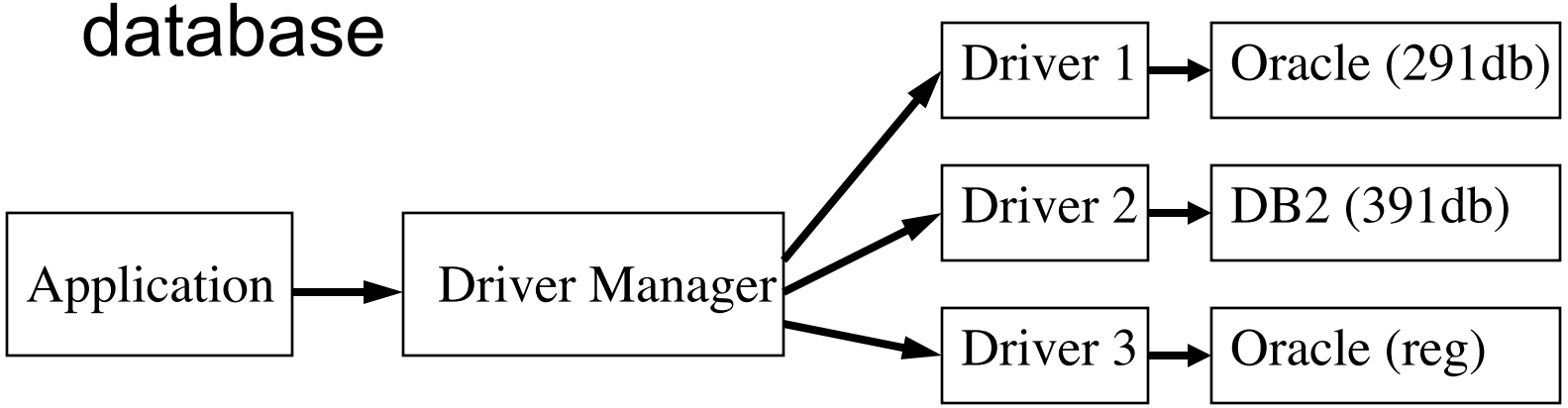
Statement-level Interface (Summary)

- 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



- We discuss JDBC and Callback for accessing our databases in Java and C

JDBC

- A Java API (a set of function calls) to communicate with SQL engines
- Supports database queries and updates as well as metadata retrievals (e.g. names of tables and columns)

Steps

- Open a connection

```
Connection conn = DriverManager.getConnection(url);
```

- Create a statement

```
Statement stmt = conn.createStatement();
```

- Execute the queries and fetch the results

```
ResultSet rs = stmt.executeQuery("select id, name, phone from emp;")
// process the result, one row at a time
```

- Handle errors (exception handling)

```
try {…}
catch (SQLException e) {
        System.out.println("SQLException : " + e);
}
```

A Simple Program (P1.java)

```
import java.sql.*;

public class P1 {

  public static void prog1 () {

    String driverName = "org.sqlite.JDBC";
    String url ="jdbc:sqlite:/Users/drafiei/Courses/291/code/jdbc/291dbfile.db";
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;

    try {
        Class.forName(driverName);
    } catch (ClassNotFoundException e) {
       System.out.println("ClassNotFoundException : " + e);
    }
```

```
    try {
      con = DriverManager.getConnection(url);
      stmt = con.createStatement();
      rs = stmt.executeQuery("select id, name, phone from emp");

      while(rs.next()) {
        System.out.println(rs.getInt(1) + "   " +
          rs.getString(2) + "   " +
          rs.getString(3));
      }
      stmt.close();
      con.close();
    }
    catch (SQLException sqle) {
        System.out.println("SQLException : " + sqle);
    }
  }

   public static void main (String[] args) { prog1();}
}
```

Compile and Run

- Create table emp(id, name, phone) with some tuples and store it in 291dbfile.db
- Have sqlite-jdbc in class path
- Compile and run

```
javac -cp "../sqlite-jdbc-3.27.2.1.jar" P1.java
java -cp "../sqlite-jdbc-3.27.2.1.jar:." P1
```

Include the current directory that has the class file

Executing a Query

import java.sql.*; // *imports all classes in package* java.sql
String driverName = "org.sqlite.JDBC";
String db_url = "jdbc:sqlite:/Users/drafiei/291dbfile.db";

//String driverName = "oracle.jdbc.driver.OracleDriver";
//String db_url = "jdbc:oracle:thin:@gwynne.cs.ualberta.ca:1521:CRS";

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 stmt = con.CreateStatement ();

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

Executing a Query

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

ResultSet rs = stmt.executeQuery (query);

- *Creates a result set object,* rs.
- *Prepares and executes the query.*
- *Stores the result set produced by execution in* rs *(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

Scrollable Result Set

```
stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                           ResultSet.CONCUR_READ_ONLY);
rs = stmt.executeQuery("select id, name, phone from emp");

// playing with a scrolable Result Set

rs.absolute(3);  // jump to Row 3

// Traverse the resultset from bottom
rs.afterLast();
while(rs.previous()) {
System.out.println(rs.getInt(1) + "   " +
  rs.getString(2) + "   " +
  rs.getString(3));
}
```

ResultSet Type Values

- TYPE_FORWARD_ONLY
- TYPE_SCROLL_INSENSITIVE
- TYPE_SCROLL_SENSITIVE

SQLite only supports TYPE_FORWARD_ONLY

Updateable Result Set

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

- Any result set type can be declared *read-only* (CONCUR_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/deleted and a new row can be inserted, causing changes in base table

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

SQLite only supports CONCUR_READ_ONLY

Accessing Metadata

- Metadata includes table names, column counts, column names and types, etc.

```
rs = stmt.executeQuery("select * from emp");

// Access metadata
ResultSetMetaData rsmd = rs.getMetaData();
for (int i = 1; i <= rsmd.getColumnCount(); i++) {
    System.out.println(rsmd.getColumnName(i) + " : " +
                       rsmd.getColumnTypeName(i));
}
```

Controlling Transactions

- Each SQL statement is treated as a transaction and is committed automatically
  - Not always a good strategy (?)
- Alternative
  - Turn off auto commit and commit/rollback explicitly

```
con = DriverManager.getConnection(url);
con.setAutoCommit(false);

String sql = "INSERT INTO emp(id,name,phone) VALUES(?,?,?)";
PreparedStatement pstmt = con.prepareStatement(sql);
pstmt.setInt(1, 0); pstmt.setString(2, "Bob"); pstmt.setString(3, "780-000-0000");
pstmt.executeUpdate();

con.commit();   // com.rollback();
```

JDBC (Summary)

- More flexible than Statement Level Interface
- Schema and table names may not be known in advance
- Scrollable and updatable result set
  (though sqlite is limited on this aspect)
- Can connect to any number of databases

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 (no SQLite support)
  - 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.