Introduction to Database Programming

Presented by: Yunliang Jiang

2010.10.14

DB Programming

- SQL is a very high-level language.
- **Not** intended for general-purpose computations.
- Solutions:
  - Outside DBMS: use SQL together with general-purpose programming languages.
  - Inside DBMS: augment SQL with constructs from general-purpose programming languages.

All these methods follow the same basic paradigm

1. Connect to a DB server.
2. Say what database you want to use.
3. Assemble a string containing an SQL statement.
4. Get the DBMS to prepare a plan for executing the statement.
5. Execute the statement.
6. Extract the results into variables in the local programming language.

Overview

- Outside
  - API Approach:
    - Vendor specific libraries [80's-]
      - MySQL API for PHP
    - Open interfaces [90's -]
      - JDBC, ODBC
  - Embedded SQL [70's-]
    - Embedded SQL for C/C++.
    - Not widely used.
- Inside
  - Stored procedures/functions: [80's-]

API Approach

- Programmer uses a library of functions or classes and call them as part of an ordinary C or Java program.
- SQL commands are sent to the DBMS at runtime.
- These API's are based on an standard called SQL/CLI = "Call-Level Interface."

Database specific API

- Designed and implemented by the DBMS vendor for a specific programming language.
- We go over MySQL API for PHP.

Connection

- mysql_connect opens a connection to the DBMS.
- It gets the DBMS and login information and returns a *connection* resource. The *connection* resource is used in future calls.
- mysql_select_db selects the desired database
- mysql_close closes the connection at the end. It is called automatically at the end of the script.

Executing Statements

- mysql_query(S, C) causes the SQL statement represented by S to be executed on connection C.
- We can detect DBMS initiated errors using mysql_error().
- mysql_query returns a handle to the query result set.
  - It returns TRUE/FALSE for DELETE, UPDATE, and INSERT statements.

Fetching Tuples

- When the SQL statement executed is a query, we need to fetch the tuples of the result.
- mysql_fetch_array(H) gets the tuples from the result set *H*.
- mysql_free_result(H) frees the result set. It is called automatically at the end of the script

Fetching Tuples

- Random access fetching by mysql_data_seek.
- More functions at:
  - http://us.php.net/mysql

Open interface: JDBC

- Database specific API makes the program dependent to one DBMS.
- Open interfaces solve this problem:
  - Designed by a third party like the creator of the language.
  - Implemented by DBMS vendors.
  - Used by DB programmers.
- Programmer do not have to change their code to work with another DBMS.
- http://www.jdbc-tutorial.com/

Connections and Statements

- The same progression from connections to statements that we saw in CLI appears in JDBC.
- A *connection object* is obtained from the environment in a somewhat implementation-dependent way.
- We'll start by assuming we have myCon, a connection object.

Statements

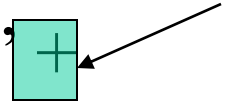
- JDBC provides two classes:
  1. Statement = an object that can accept a string that is an SQL statement and can execute such a string.
  2. PreparedStatement = an object that has an associated SQL statement ready to execute.
- Why? Performance
  - PreparedStatesment SQL command is stored in DBMS after the first call.

Creating Statements

- The Connection class has methods to create Statements and PreparedStatements.

```
Statement stat1 = myCon.createStatement();
PreparedStatement stat2 =
    myCon.prepareStatement(
        "SELECT Address FROM Supplier" +
        "WHERE Suuplier_Name = 'John Smith'"
    );
```

Java trick: + concatenates strings.



Executing SQL Statements

- Programmer handles errors using SQLException class.
- JDBC distinguishes queries from modifications, which it calls “updates.”
- Statement and PreparedStatement each have methods executeQuery and executeUpdate.
  - For Statements, these methods have one argument: the query or modification to be executed.
  - For PreparedStatements: no argument.

Example: Update

- stat1 is a Statement.
- We can use it to insert a tuple as:

```
stat1.executeUpdate(
   “INSERT INTO Supplier” +
   “VALUES(‘S4’, ‘Mary’, ‘12 Goodwin St.’)”
);
```

Example: Query

- stat2 is a PreparedStatement holding the query "SELECT Address FROM Supplier WHERE Supplier_Name = 'John Smith'".
- executeQuery returns an object of class ResultSet --- we'll examine it later.
- The query:

```
ResultSet Menu = stat2.executeQuery();
```

Accessing the ResultSet

- An object of type ResultSet is something like a cursor.
- Method Next() advances the “cursor” to the next tuple.
  - The first time Next() is applied, it gets the first tuple.
  - If there are no more tuples, Next() returns the value FALSE.

Accessing Components of Tuples

- When a ResultSet is referring to a tuple, we can get the components of that tuple by applying certain methods to the ResultSet.
- Method get*X* (*i*), where *X* is some type, and i is the component number, returns the value of that component.
  - The value must have type *X*.

Example: Accessing Components

- List is the ResultSet for the query “SELECT Address FROM Supplier WHERE Supplier_Name = ‘John Smith’”.
- Access the address from each tuple by:

```
while ( List.Next() ) {
  theAddress = List.getString(1);
     /* do something */
}
```

- More: http://java.sun.com/products/jdbc/overview.html

Programming Inside the DBMS

Stored Procedures

- An extension to SQL, called SQL/PSM, or "persistent, stored modules," allows us to store procedures as database schema elements.
- The programming style is a mixture of conventional statements (if, while, etc.) and SQL.
- Lets us do things we cannot do in SQL alone.
- Why? Performance.
- They are harder to develop and maintain.

Basic PSM Format

CREATE PROCEDURE <name> (

<parameter list>)

<optional local declarations>

<body>;

- Function alternative:

CREATE FUNCTION <name> (

<parameter list>) RETURNS <type>

Example: compute square footage of house lot; then you can query on it

Parameters in PSM

- Unlike the usual name-type pairs in languages like C, PSM uses mode-name-type triples, where the *mode* can be:
  - IN = procedure uses value, does not change value.
  - OUT = procedure changes, does not use.
  - INOUT = both.

Example: Stored Procedure

- Let's write a procedure that takes three arguments *i*, *n*, and *a*, and adds a tuple to Supplier that has Suuplier_ID = *i*, Suplier_Name = *n*, and Address = *a*.
  - Used to add new Supplier more easily.

The Procedure

```
CREATE PROCEDURE newSup (
    IN  i       VARCHAR(20),
    IN  n       VARCHAR(50),
    IN a        VARCHAR(100)
)
INSERT INTO Supplier
VALUES(i, n, a);
```

Parameters are all read-only, not changed

The body --- a single insertion

Invoking Procedures

- Use SQL/PSM statement CALL, with the name of the desired procedure and arguments.
- Example:

  CALL newSup('s5', 'Bob','1 Main St.);

- Functions used in SQL expressions where a value of their return type is appropriate.

Types of PSM statements -- 1

- RETURN <expression> sets the return value of a function.
  - Unlike C, etc., RETURN *does not* terminate function execution.
- DECLARE <name> <type> used to declare local variables.
- BEGIN . . . END for groups of statements.
  - Separate by semicolons.

Types of PSM Statements -- 2

- Assignment statements:

  SET <variable> = <expression>;

  – Example: SET b = ’Bud’;

- Statement labels: give a statement a label by prefixing a name and a colon.

Example: IF

- Let's rate suppliers by how many parts they have
  - <=0 parts: 'bad'.
  - < 2 parts: 'average'.
  - >=2 parts: 'good'.
- Function Rate(s) rates supplier s.

Example: IF (continued)

```
CREATE FUNCTION Rate (IN b VARCHAR(50) )
        RETURNS CHAR(10)
        DECLARE pCnt INTEGER;
    BEGIN
        SET pCnt = (SELECT COUNT(*) FROM Supplier,
                    Catalog WHERE Supplier.Supplier_ID =
                    Catalog.Supplier_ID and
                    Supplier_Name= b);
        IF pCnt < 0 THEN RETURN 'bad'
        ELSEIF pCnt < 2 THEN RETURN 'average'
        ELSE RETURN 'good'
        END IF;
    END;
```

Number of the parts of the supplier b

Nested IF statement

Return occurs here, not at one of the RETURN statements

Example: Exiting a Loop

```
loop1: LOOP
        . . .
        LEAVE loop1;
        . . .
END LOOP;
```

If this statement is executed . . .

Control winds up here

Other Loop Forms

- WHILE <condition>

  DO <statements>

  END WHILE;
- REPEAT <statements>

  UNTIL <condition>

  END REPEAT;

Queries

- General SELECT-FROM-WHERE queries are *not* permitted in PSM.
- There are three ways to get the effect of a query:
  1. Queries producing one value can be the expression in an assignment.
  2. Single-row SELECT . . . INTO.
  3. Cursors.

Example: Assignment/Query

- If *a* is a local variable and Supplier(Supplier_ID, Supplier_Name, Address) the usual relation, we can get the address of ‘John Smith’ by:

```
SET a = (SELECT Address FROM Supplier
         WHERE Supplier_Name = 'John
         Smith');
```

SELECT . . . INTO

- An equivalent way to get the value of a query that is guaranteed to return a single tuple is by placing INTO <variable> after the SELECT clause.
- Example:

```
SELECT Address INTO a FROM
Supplier WHERE Supplier_Name =
'John Smith';
```

Cursors

- A *cursor* is essentially a tuple-variable that ranges over all tuples in the result of some query.
- Declare a cursor *c* by:

DECLARE c CURSOR FOR <query>;

Opening and Closing Cursors

- To use cursor *c*, we must issue the command:

  OPEN c;

  – The query of *c* is evaluated, and *c* is set to point to the first tuple of the result.

- When finished with *c*, issue command:

  CLOSE c;

Fetching Tuples From a Cursor

- To get the next tuple from cursor c, issue command:

  FETCH FROM c INTO x1, x2,…,x*n* ;

- The *x* 's are a list of variables, one for each component of the tuples referred to by *c*.
- c is moved automatically to the next tuple.

Breaking Cursor Loops -- 1

- The usual way to use a cursor is to create a loop with a FETCH statement, and do something with each tuple fetched.
- A tricky point is how we get out of the loop when the cursor has no more tuples to deliver.

Breaking Cursor Loops -- 2

- Each SQL operation returns a *status*, which is a 5-digit number.
  - For example, 00000 = "Everything OK," and 02000 = "Failed to find a tuple."
- In PSM, we can get the value of the status in a variable called SQLSTATE.

Breaking Cursor Loops -- 3

- We may declare a condition, which is a boolean variable that is true if and only if SQLSTATE has a particular value.
- Example: We can declare condition NotFound to represent 02000 by:

```
DECLARE NotFound CONDITION FOR
    SQLSTATE '02000';
```

Breaking Cursor Loops -- 4

- The structure of a cursor loop is thus:

```
cursorLoop: LOOP
    …
    FETCH c INTO … ;
    IF NotFound THEN LEAVE cursorLoop;
    END IF;
    …
END LOOP;
```

Example: Cursor

- Let's write a procedure that examines Supplier and Catalog and raises by $1 the cost of all parts in the 'John Smith' inventory that are under $10.
  - Yes, we could write this as an UPDATE, but the details are instructive anyway.

The Needed Declarations

```
CREATE PROCEDURE raiseCost( )
  DECLARE theID VARCHAR(50);
  DECLARE theCost REAL;
  DECLARE NotFound CONDITION FOR
      SQLSTATE '02000';
  DECLARE c CURSOR FOR
    (SELECT Supplier.Supplier_ID, Cost FROM
    Supplier, Catalog WHERE Supplier.Supplier_ID =
    Catalog.Supplier_ID and Supplier_Name = 'John
    Smith');
```

Used to hold ID-cost pairs when fetching through cursor c

Returns John's parts costs

The Procedure Body

```
BEGIN
    OPEN c;
    menuLoop: LOOP
        FETCH c INTO theID, theCost;
        IF NotFound THEN LEAVE menuLoop END IF;
        IF theCost < 10.00 THEN
            UPDATE Catalog SET Cost = theCost+1.00
            WHERE Supplier_ID = theID;
        END IF;
    END LOOP;
    CLOSE c;
END;
```

Check if the recent FETCH failed to get a tuple

If John charges less than $10 for the part raise it's cost by $1.

Further Readings

- http://dev.mysql.com/doc/refman/5.0/en/stored-routines.html
- http://www.oracle.com/technology/tech/pl_sql/index.html

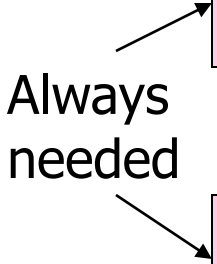
Embedded SQL

Embedded SQL

- *A* standard for combining SQL with different languages.
- Key idea: Use a preprocessor to turn SQL statements into procedure calls that fit with the host-language code surrounding.
- All embedded SQL statements begin with EXEC SQL, so the preprocessor can find them easily.

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: 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.

Example: C Plus SQL

```
EXEC SQL BEGIN DECLARE SECTION;
   char theID[21];
   float theCost;
EXEC SQL END DECLARE SECTION;
   /* obtain values for theID and theCost */
EXEC SQL SELECT Cost INTO :theCost
   FROM Catalog
   WHERE Supplier_ID = theID;
   /* do something with theCost */
```

Note 21-char arrays needed for 20 chars + endmarker

SELECT-INTO just like PSM

Embedded Queries

- Embedded SQL has the same limitations as PSM regarding 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 list of Supplier's with the name 'John Smith'.
- A cursor will visit each Supplier tuple that has Supplier_Name = 'John Smith'.

Example – 2 (Declarations)

```
EXEC SQL BEGIN DECLARE SECTION;
   char theID[21]; char theAddress[151];
EXEC SQL END DECLARE SECTION;
EXEC SQL DECLARE c CURSOR FOR
   SELECT theID, theAddress FROM Supplier
   WHERE Supplier_Name = 'John Smith';
```

The cursor declaration goes outside the declare-section

Example – 3 (Executable)

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

The C style of breaking loops

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>;

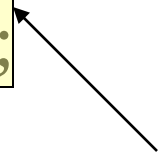
- Executing a query:

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;
}
```



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;
}
```

Further Readings

http://infolab.stanford.edu/~ullman/fcdb/oracle/or-proc.html