Chapter 5: Advanced SQL

- Accessing SQL From a Programming Language
  - Dynamic SQL
    - JDBC and ODBC
  - Embedded SQL
- SQL Data Types and Schemas
- Functions and Procedural Constructs
- Triggers

Accessing SQL from a general Purpose programming language

Two main reasons:

1. Not all queries can be expressed in SQL
2. Nondeclarative Actions (e.g., printing, output formatting, interactive actions, ...

SQL commands can be called from within a host language (e.g., C++ or Java) program.

- SQL statements can refer to host variables (including special variables used to return status).
- Must include a statement to *connect* to the right database.

Two Main approaches:

- **Dynamic SQL:** allows program to construct an SQL query as a char. String at run time, submit the query and retrieve results tuple-at-time into program variables.
- **Embedded SQL**: SQL statements are identified at compile time by preprocessor, complied and optimized by database system, and replaced by appropriate code and function calls before programming language compiler is invoked.

Impedance mismatch:

SQL relations are (multi-) sets of records, with no *a priori* bound on the number of records. Sets are not supported cleanly in procedural programming languages

- SQL supports a mechanism called a *cursor* to handle this.

Embedded SQL

- Approach: Embed SQL in the host language.
  - An DBMS-specific preprocessor converts the SQL statements into special API calls.
  - Then a regular compiler is used to compile the code.

- Language constructs:
  - Connecting to a database:
    EXEC SQL CONNECT
  - Declaring variables:
    EXEC SQL BEGIN (END) DECLARE SECTION
  - Statements:
    EXEC SQL Statement;

Embedded SQL: Variables

```
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20];
long c_sid;
short c_rating;
float c_age;
EXEC SQL END DECLARE SECTION
```

Two special “error” variables:

- SQLCODE (long, is negative if an error has occurred)
- SQLSTATE (char[6], predefined codes for common errors)

Problem 1:Data types in SQL must be recognized by host language, and vice versa.
Sol’n: Use casting (of data values of one type into values of the other one).

Problem 2: DB query language: set-oriented, and host langauge: value oriented.
Sol’n: Use cursors.

Cursors

- Can declare a cursor on a relation or query statement (which generates a relation).
- Can *open* a cursor, and repeatedly *fetch* a tuple then *move* the cursor, until all tuples have been retrieved.
  - Can use a special clause, called ORDER BY, in queries that are accessed through a cursor, to control the order in which tuples are returned.
  - Fields in ORDER BY clause must also appear in SELECT clause.
  - The ORDER BY clause, which orders answer tuples, is *only* allowed in the context of a cursor.
- Can also modify/delete tuple pointed to by a

Cursor that gets names of students who’ve enrolled in EECS 341, in alphabetical order

```
EXEC SQL DECLARE sinfo CURSOR FOR
    SELECT  S.sname
    FROM  Student S, Enrolled E
    WHERE  S.sid=E.sid AND E.cid=“EECS 341”
    ORDER BY  S.sname

OPEN sinfo;
```

Embedded SQL (Cont.)

- The **open** statement causes the query to be evaluated

  EXEC SQL **open** *c* END_EXEC

- The **fetch** statement causes the values of one tuple in the query result to be placed on host language variables.

  EXEC SQL **fetch** *c* **into** :*si*, *:sn* END_EXEC

  Repeated calls to **fetch** get successive tuples in the query result

- A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to '02000' to indicate no more data is available

- The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

  EXEC SQL **close** *c* END_EXEC

  Note: above details vary with language. For example, the Java embedding defines Java iterators to step through result tuples.

Embedding SQL in C: An Example

```
char SQLSTATE[6];
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20]; char c_major; short c_year;
EXEC SQL END DECLARE SECTION
c_major = 'CS';
EXEC SQL DECLARE sinfo CURSOR FOR
  SELECT S.sname, S.year  FROM Students S
  WHERE S.major = :c_major  // prefix " : " for variables defined
  ORDER BY S.sname;                 //   in host program
EXEC SQL OPEN sinfo
do {
  EXEC SQL FETCH sinfo INTO :c_sname, :c_year;
  printf( c_sname, c_year);
} while (SQLSTATE != '02000');
EXEC SQL CLOSE sinfo;
```

Example Query

- From within a host language, find the ID and name of students who have completed more than the number of credits stored in variable credit_amount.

  - Specify the query in SQL and declare a *cursor* for it

```
EXEC SQL
declare c cursor for
select ID, name
from student
where tot_cred > :credit_amount
END_EXEC
```

Updates Through Cursors

- Can update tuples fetched by cursor by declaring that the cursor is for update

```
declare c cursor for
  select *
  from instructor
  where dept_name = 'Music'
for update
```

- To update tuple at the current location of cursor *c*

```
update instructor
set salary = salary + 100
where current of c
```

Dynamic SQL

- SQL query strings are not always known at compile time (e.g., spreadsheet, graphical DBMS frontend): Allows construction of SQL statements on-the-fly.

- **Example:**

```
char c_sqlstring[]=
  {“DELETE FROM Students WHERE year = “freshmen”};
EXEC SQL PREPARE readytogo FROM :c_sqlstring;
EXEC SQL EXECUTE readytogo;
```

Database APIs: Alternative to embedding

Rather than modify compiler, add library with database calls (API).

- Special standardized interface: procedures/objects
- Passes SQL strings from language; presents result sets in a language-friendly way.
- ODBC (open database connectivity)
- Sun's *JDBC:* Java API
- Supposedly DBMS-neutral
  - a "driver" traps the calls and translates them into DBMS-specific code.
  - database can be across a network.

JDBC: Architecture

- Four architectural components:
  - Application (initiates and terminates connections, submits SQL statements).
  - Driver manager (loads JDBC driver).
  - Driver (connects to data source, transmits requests and returns/translates results and error codes).
  - Data source (processes SQL statements).

JDBC Classes and Interfaces

Steps to submit a database query:

1. Load the JDBC driver
2. Connect to the data source
3. Execute SQL statements

Executing SQL Statements in JDBC

- Three different ways of executing SQL statements:
  - Statement (both static and dynamic SQL statements)
  - PreparedStatement (semi-static SQL statements)
  - CallableStatement (stored procedures)

- PreparedStatement class:
  - Precompiled, parameterized SQL statements
  - Structure is fixed.
  - Values of parameters are determined at run-time.

ResultSets in JDBC

- PreparedStatement.executeUpdate only returns the number of affected records.
- PreparedStatement.executeQuery returns data, encapsulated in a ResultSet object (a cursor).

```
ResultSet rs=pstmt.executeQuery(sql);
// rs is now a cursor
While (rs.next()) {
  // process the data
}
```

ResultSets (Contd.)

A ResultSet is a very powerful cursor:

- previous(): moves one row back.
- absolute(int num): moves to the row with the specified number.
- relative (int num): moves forward or backward.
- first() and last()

JDBC: Exceptions and Warnings

- Most of java.sql can throw a SQLException if an error occurs.
- SQLWarning is a subclass of SQLException; not as severe (they are not thrown and their existence has to be explicitly tested)

SQL Procedures

- The *dept_count* function could instead be written as procedure:

  create procedure *dept_count_proc* (**in** *dept_name* **varchar**(20),
  out *d_count* **integer**)
  begin
  select count(*) **into** *d_count*
  from *instructor*
  where *instructor.dept_name = dept_count_proc.dept_name*
  end

- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the **call** statement.

  declare *d_count* **integer**;
  call *dept_count_proc*(‘Physics’, *d_count*);