Database Application Development

Chapter 6.1-6.4

"Mommy wants you to know where your food comes from."

Overview

Concepts covered in this lecture:

- SQL in application code
- Embedded SQL
- Cursors
- Dynamic SQL
- sqlite3 in Python

Justification for access to databases via programming languages:

- SQL is a direct query language; as such, it has limitations.
- Standard programming languages:
  - Complex computational processing of the data.
  - Specialized user interfaces.
  - Logistics and decision making
  - Access to more than one database at a time.

SQL in Application Code

- Most often SQL commands are called from within a host language (e.g., Java or Python) program.
  - SQL statements can reference and modify host language variables (including special variables used to return results and status).
  - Must include an API to *connect* to and *issue queries* to the right database.

SQL in Application Code (Contd.)

Impedance mismatch:

- Philosophical differences in the data models used by SQL and programming languages
- SQL relations are (multi-) sets of tuples, with no *a priori* bound on the number of tuples.
- No such data structure exist in traditional procedural programming languages such as C++. (Though now: Python)
- SQL language interfaces often support a mechanism called a *cursor* to handle this.

Desirable features of such systems:

- Ease of use.
- Conformance to standards for existing programming languages, database query languages, and development environments.
- Interoperability: the ability to use a common interface to diverse database systems on different operating systems

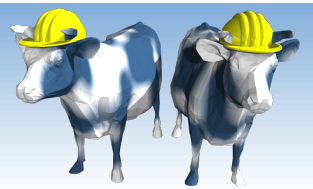
Vendor specific solutions

- Oracle PL/SQL: A proprietary PL/1-like language which supports the execution of SQL queries:
- Advantages:
  - Many Oracle-specific features, not common to other systems, are supported.
  - Performance may be optimized for Oracle-based systems.
- Disadvantages:
  - Ties the applications to a specific DBMS.
  - The application programmer must depend upon the vendor for the application development environment.
  - It may not be available for all platforms.

Vendor Independent solutions based on SQL

Three basic strategies:

- Embed SQL in the host language (Embedded SQL, SQLJ)
  - SQL code appears inline with other host-language code
  - Calls are resolved at compile tiome
- SQL call-level interfaces (Dynamic SQL)
  - Wrapper functions that pass strings from the host language to a separate interpreted SQL process
- SQL modules or libraries



Embedded SQL

- Approach: Embed SQL in the host language.
  - A 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

- There is a need for the host language to share variable with the database's SQL interface:

```
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)

Disadvantages:

- Directives must be preprocessed, with subtle implications for code elsewhere
- It is a real pain to debug preprocessed programs.
- The use of a program-development environment is compromised substantially.
- The preprocessor must be vendor and platform specific.

Dynamic SQL

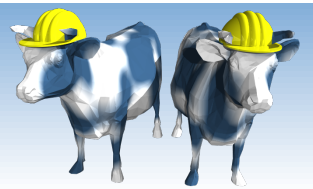
- SQL query strings are not always known at compile time (e.g., spreadsheet, graphical DBMS frontend): Allow construction of SQL statements on-the-fly

- Example:

```
char c_sqlstring[]=
    {"DELETE FROM Sailors WHERE rating>5"};
EXEC SQL PREPARE readytogo FROM :c_sqlstring;
EXEC SQL EXECUTE readytogo;
```

SQL Modules

- In the module approach, invocations to SQL are made via libraries of procedures , rather than via preprocessing
- Special standardized interface: procedures/objects
- Pass SQL strings from language, presents result sets in a language-friendly way
- Supposedly DBMS-neutral
  - a "driver" traps the calls and translates them into DBMS-specific code
  - database can be across a network



Example module based

- Python's built-in SQLite package
  - Add-ons for
    - MySQL (MySQL for Python),
    - Oracle (Oracle+Python, cx_Oracle)
    - Postgres (PostgreSQL)
    - etc.
- Sun's *JDBC:* Java API
- Part of the java.sql package

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.
- In some cases, you can also modify/delete tuple pointed to by a cursor, and changes are reflected in the database

Get names of sailors who've reserved a red boat, by rating in alphabetical order

- First, one more SQL feature

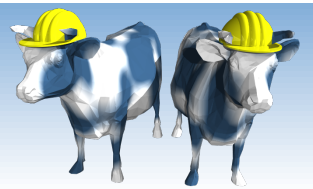
```
SELECT  S.sname, S.rating
FROM  Sailors S, Boats B, Reserves R
WHERE  S.sid=R.sid AND R.bid=B.bid AND B.color='red'
ORDER BY  S.rating DESC, S.sname ASC
```

- Note that the ORDER BY clause determines the order which query results are returned
- Can use multiple attribute names to resolve ties
- Optional ASC or DESC keyword after attribute for ascending or descending order respectively

Verdict on SQL Modules

- Advantages over embedded SQL:
  - Cleaner separation of SQL from the host programming language.
  - Debugging is much more straightforward, since no preprocessor is involved.

- Disadvantages:
  - The module libraries are specific to the programming language and DBMS environment. Thus, portability is somewhat compromised.



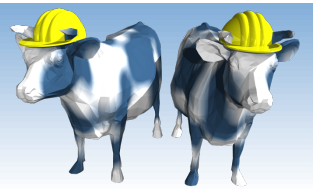
Python and SQL Data Types

Python type	SQLite type
None	NULL
int	INTEGER
long	INTEGER
float	REAL
str (UTF8-encoded)	TEXT
unicode	TEXT
buffer	BLOB

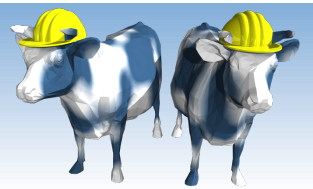
SQLite type conversions to Python

SQLite type	Python type
NULL	None
INTEGER	int or long, depending on size
REAL	float
TEXT	depends on text_factory, unicode by default
BLOB	buffer

Embedding SQL in Python



```
import sqlite3
if __name__ == '__main__':
    db = sqlite3.connect("sailors.db")
    cursor = db.cursor()

    cursor.execute("""SELECT s.sname, b.bname, r.day
                      FROM Sailors s, Reserves r, Boats b
                      WHERE s.sid=r.sid AND r.bid=b.bid
                        AND b.color='red'
                      ORDER BY s.sname""")

    print "     Name        Boat         Date"
    for row in cursor:
        print "%12s %12s %10s" % row

    db.close()
```



More Involved Example

- Increase after three or more reservations

```
import sqlite3
if __name__ == '__main__':
    db = sqlite3.connect("sailors.db")
    cursor = db.cursor()
    print "BEFORE"
    cursor.execute("SELECT * FROM Sailors")
    for row in cursor:
        print row

    cursor.execute("""SELECT s.sid, COUNT(r.bid) AS reservations
                      FROM Sailors s, Reserves r
                      WHERE s.sid=r.sid
                      GROUP BY s.sid
                      HAVING s.rating < 10""")

     for row in cursor.fetchall():
        if (row[1] > 2):
            cursor.execute("""UPDATE Sailors
                              SET rating = rating + 1
                              WHERE sid=%d""" % row[0])

    print "AFTER"
    cursor.execute("SELECT * FROM Sailors")
    for row in cursor:
        print row
    db.close()
```

SQL could do more or less of the work in this simple example

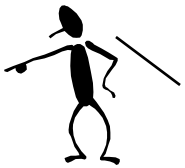
Where Python and SQL meet

- UGLY inter-language semantics
  - Within SQL we can reference a relation's attributes by its field name
  - From the cursor interface we only see a tuple in which attributes are indexed by position
  - Can be a maintenance nightmare
- Solution "Row-factories"
  - Allows you to remap each relation to a local Python data structure (Object, dictionary, array, etc.)
  - Built-in "dictionary-based" row factory

With a Row-Factory

```
import sqlite3

if __name__ == '__main__':
    db = sqlite3.connect("sailors.db")
    db.row_factory = sqlite3.Row
    cursor = db.cursor()

    cursor.execute("""SELECT s.sid, COUNT(r.bid) as reservations
                      FROM Sailors s, Reserves r
                      WHERE s.sid=r.sid
                      GROUP BY s.sid
                      HAVING s.rating < 10""")

    for row in cursor.fetchall():
        if (row['reservations'] > 2):
            cursor.execute("""UPDATE Sailors
                              SET rating = rating + 1
                              WHERE sid=%d""" % row['sid'])

    db.commit()
    db.close()
```

Must come before dependent cursor



Must "commit" to make INSERTs and/or UPDATEs persistant

Other SQLite in Python Features

- Alternatives to iterating over cursor
  - Fetch the next tuple:

    ```
    tvar = cursor.fetchone()
    ```

  - Fetch N tuples into a list:

    ```
    lvar = cursor.fetchmany(N)
    ```

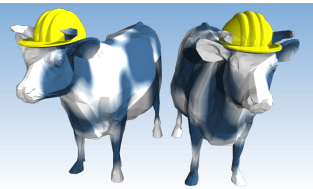
  - Fetch all tuples into a list:

    ```
    lvar = cursor.fetchall()
    ```

- Alternative execution statement
  - Repeat the same command over an iterator
    `cursor.executemany("SQL Statement", args)`
  - Execute a list of ';' separted commands
    `cursor.executescript("SQL Statements;")`

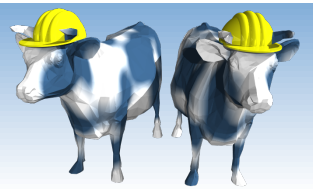
Substitution

- Usually your SQL operations will need to use values from Python variables. You shouldn't assemble your query using Python's string formatters because doing so is insecure; it makes your program vulnerable to an SQL injection attack.
- Instead, use the DB-API's parameter substitution. Put '?' as a placeholder wherever you want to use a value, and then provide a tuple of values as the second argument to the cursor's execute() method.



With a Row-Factory

```
import sqlite3

if __name__ == '__main__':
    db = sqlite3.connect("sailors.db")
    db.row_factory = sqlite3.Row
     cursor = db.cursor()

     cursor.execute("""SELECT s.sid, COUNT(r.bid) as reservations
                       FROM Sailors s, Reserves r
                       WHERE s.sid=r.sid
                       GROUP BY s.sid
                       HAVING s.rating < 10""")

     for row in cursor.fetchall():
         if (row['reservations'] > 2):
             cursor.execute("""UPDATE Sailors
                               SET rating = rating + 1
                               WHERE sid=?""", (row['sid'],))
     db.commit()
     db.close()
```



Extracting the dB's Schema

```
[~/Courses/Comp521_S10/Stuff]$ python
Python 2.6.4 (r264:75706, Nov 12 2009, 00:21:44)
[GCC 4.2.1 (Apple Inc. build 5646) (dot 1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import sqlite3
>>> db = sqlite3.connect('Sailors.db')
>>> cursor = db.cursor()
>>> cursor.execute("SELECT * FROM sqlite_master WHERE type='table'")
<sqlite3.Cursor object at 0x100430920>
>>> for row in cursor:
...     print row
...
(u'table', u'Sailors', u'Sailors', 2, u'CREATE TABLE Sailors( sid INTEGER,
                                                sname STRING,
                                                rating INTEGER,
                                                age REAL)')
(u'table', u'Boats', u'Boats', 3, u'CREATE TABLE Boats(   bid INTEGER,
                                                bname STRING,
                                                color STRING)')
(u'table', u'Reserves', u'Reserves', 4, u'CREATE TABLE Reserves(sid INTEGER,
                                                bid INTEGER,
                                               day DATE)')
>>>
```

Next Time

- JDBC approach from embedding SQL
- Extra levels of indirection to translate between between a uniform database API and alternate DBMS backends