Database Application Development

The merge is official

PS #2 now applies to both sections, stay tuned for an announcement WRT its due date
Comp 521 Changes...

❖ There will be no midterm next week in section 001

5 - Problem sets (lowest dropped) 30% 20%
N - In-class exercise/worksheets 10% 10%
Midterm 30% 30%
Final Exam 30% 40%

❖ I will retrofit my problem sets so that they are compatible with Prof. Bishop's validators and autograders.
❖ Expect a new version of iSQL.py. But this should not keep you from starting the problem set
❖ We will continue with section 001's Piazza, and section 002 will be added
❖ I hope to convert my "exercises" into "worksheets"
Onboarding

If you were here last Thursday, you should have
❖ A course website login

Let’s try each.
First goto https://csbio.unc.edu/mcmillan/

Click Here
Course website login

Username: ONYEN
Password: ********

Enter your ONYEN as your username, and your PID as your password.

Your login should then show up as “Verified”
Next press “Continue”; you should then see “Setup” as a menu option. Press it.
You should probably change your password, but don’t forget it.

Here’s the link to active in-class exercises and worksheets.

Comp521F19 Problem Sets and Exams:

Comp521F19 Exercises:

Exercises:
fan8 has submitted 1 of 1 exercises

Exercise01:
#URL=https://docs.google.com/forms/d/e/1FAIpQLSdb2Xo1ZdUmSVjmrQ8uohBqnASKiULTBNiunZQ9EFk5ndt0_0/viewform?usp=sf_link

You should probably change your password, but don’t forget it.
Using databases within programs

- Often need to access databases from programming languages
  - as a file alternative
  - as shared data
  - as persistent state

- 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 multiple databases
SQL in Application Code

- Most often SQL commands are called from within a host language (e.g., Java or Python) program.
  - SQL statements need to reference and modify *host language variables* (with special variables used to return results and status).
  - Generally, an Application Programming Interface (API) is used to *connect to, issue queries, modify, and update* databases.
SQL in Application Code (Contd.)

Impedance mismatch:
- Differences in the data models used by SQL and programming languages
- SQL relations are (multi-) sets of tuples, with no \textit{a priori} bound on number, length, and type.
- No such data structure exist in traditional procedural programming languages such as C++. (But Python has it!)
- SQL language interfaces often support a mechanism called a \textit{cursor} iterator.
Desirable features of SQL APIs:

❖ 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 access diverse database management 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, high performance, tight integration.
    - Advantage, overall performance can be optimized by analyzing both the queries and the surrounding program logic.

  - 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

Three basic strategies:

- Embed SQL in the host language (Embedded SQL, SQLJ)
  - SQL code appears inline with other host-language code
  - Queries are determined at compile time

- SQL call-level interfaces (Dynamic SQL)
  - Wrapper functions that pass SQL queries as strings from the host language to a separate DBMS process
  - This allows queries to be constructed "programmatically"

- 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:

  ```sql
  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)
Cursors

- Can declare a cursor on a relation or query statement that generates a relation.
- Can open a cursor, and repeatedly fetch tuples and advance the cursor as a side-effect, until all tuples have been retrieved.
- In some cases, you can also modify/delete tuple pointed to by a cursor, and changes are reflected in the database.
**Embedded Database Use**

- **Loading a table**
  
  ```sql
  EXEC SQL
  INSERT INTO Sailors
  VALUES(:c_sname, :c_sid, :c_rating, :c_age);
  ```

- **Executing a query**
  
  ```sql
  DECLARE sinfo CURSOR FOR
  SELECT S.sname, S.age
  FROM Sailors S
  WHERE S.rating > 6;

  OPEN sinfo;
  do {
      FETCH sinfo INTO :c_name, :c_age;
      /* do stuff */
      if (c_name == "dustin") {
          ageSum += c_age;
          dustinCount += 1;
      }
  } while (SQLSTATE != NO_DATA); /* NO_DATA == “02000” */
  CLOSE sinfo;
  ```
Embedded SQL 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 is “compiler 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:
  ```c
  char c_sqlstring[] = 
  {"DELETE FROM Sailors WHERE rating>5"};
  EXEC SQL PREPARE readytogo FROM :c_sqlstring;
  EXEC SQL EXECUTE readytogo;
  ```
SQL Package and Libraries

- In the package 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
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 tend to be specific to the programming language and DBMS. Thus, portability is somewhat compromised.
SQL in Python

- Python is a high-level interpreted language with dynamic types
- High-level means that is provide a rich set of data structures built-in to the language with strong abstractions from the details of their implementation
- Tuples are a built-in datatype which makes it particularly compatible with relational databases
- A SQLite API is built into Python.
# Python and SQL Data Types

<table>
<thead>
<tr>
<th>Python type</th>
<th>SQLite type</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td>NULL</td>
</tr>
<tr>
<td>int</td>
<td>INTEGER</td>
</tr>
<tr>
<td>long</td>
<td>INTEGER</td>
</tr>
<tr>
<td>float</td>
<td>REAL</td>
</tr>
<tr>
<td>str (UTF8-encoded)</td>
<td>TEXT</td>
</tr>
<tr>
<td>unicode</td>
<td>TEXT</td>
</tr>
<tr>
<td>buffer</td>
<td>BLOB</td>
</tr>
</tbody>
</table>
# SQLite type conversions to Python

<table>
<thead>
<tr>
<th>SQLite type</th>
<th>Python type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>None</td>
</tr>
<tr>
<td>INTEGER</td>
<td>int or long, depending on size</td>
</tr>
<tr>
<td>REAL</td>
<td>float</td>
</tr>
<tr>
<td>TEXT</td>
<td>depends on text_factory, unicode by default</td>
</tr>
<tr>
<td>BLOB</td>
<td>buffer</td>
</tr>
</tbody>
</table>
import sqlite3

db = sqlite3.connect("NFL.db")
cursor = db.cursor()

cursor.execute("""SELECT P.name, R.jersey, R.position
    FROM Player P, PlayedFor R, Team T
    WHERE P.pid=R.pid AND R.tid=T.tid
        AND T.mascot='chiefs' AND R.year=2019 AND R.jersey<>'--'
    ORDER BY R.jersey"""")

print("          Name        Jersey Position")
for row in sorted(cursor, key=lambda tup: int(tup[1])):
    if (int(row[1]) < 20):
        print("%20s %5s  %6s" % row)

db.close()
More Involved Example

What is then name, jersey number, age, and number of seasons played for each active quarterback (i.e. playing on a 2019 roster)?

```python
import sqlite3
import datetime

db = sqlite3.connect("NFL.db")
cursor = db.cursor()

cursor.execute("SELECT P.name, R.jersey, P.dob, MIN(R.year), T.mascot FROM Player P, PlayedFor R, Team T WHERE P.pid=R.pid AND R.tid=T.tid AND dob<>'--' AND P.pid in (SELECT pid FROM PlayedFor WHERE year=2019 AND position='QB') GROUP BY P.pid ORDER BY P.dob"")

print("          Name        Jersey  Age   Seasons    Team")
for row in cursor:
    ymd = [int(v) for v in row[2].split('-')]
    age = int(((datetime.date.today() - datetime.date(ymd[0],ymd[1],ymd[2])).days/365.25)
    seasons = datetime.date.today().year - int(row[3])
    print("%20s %5s %6d %6d %18s" % (row[0],row[1],age,seasons,row[4]))

db.close()
```
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

Increase the rating of all sailors who have made more than two reservations by one unless their rating is already 10.

```python
import sqlite3

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 AND s.rating < 10" 
               "GROUP BY s.sid")

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, DELETEs, and/or UPDATEs persistent
Other SQLite in Python Features

- Alternatives to iterating over cursor
  - Fetch the next tuple:
    
    ```python
tvar = cursor.fetchone()
    ```
  - Fetch N tuples into a list:
    
    ```python
lvar = cursor.fetchmany(N)
    ```
  - Fetch all tuples into a list:
    
    ```python
lvar = cursor.fetchall()
    ```

- Alternative execution statement
  - Repeat the same command over an iterator
    
    ```python
cursor.executemany("SQL Statement", args)
    ```
  - Execute a list of ‘;’ separated commands
    
    ```python
cursor.executescript("SQL Statements;")
    ```
Variable 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 SQL injection attacks.

❖ 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.
import sqlite3

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 + ?
            WHERE sid=?""", (value,row['sid']))

db.commit()
db.close()
Next Time

- A first look at query performance
- Building and using indices