Exam 4 Review: SQL, pymysql, and XML
CS 2316 – Fall 2011

This is a non-exhaustive list of topics to study. You will be held responsible for all readings on the course website and lecture contents even if they are not mentioned on this (non-complete) review sheet.

Basic SQL

Table name: players

playerID	name	position
21	Joe Jackson	QB
13	George Burdell	DE
1	Ron Howard	DT

- SELECT
  - To extract information that already exists in the database.
  - *SELECT* field names desired
    FROM table name
    WHERE condition
  - To select all the fields, use SELECT *
  - The WHERE clause is optional
  - Field names and table names are typically case sensitive
- INSERT
  - To insert new information into the database
  - *INSERT INTO* table name(fieldname1, fieldname2)
    VALUES (value1, value2)
  - If you will insert a value for each field, then you can use:
  - *INSERT INTO* table name
    VALUES (value1, value2, value3)
- UPDATE
  - To update an already existing database entry
  - *UPDATE* table name
    SET fieldname1 = value1, fieldname2 = value2
    WHERE condition
  - The WHERE clause is once again optional
- DELETE
  - To delete an existing entry

- *DELETE FROM* table name
  WHERE condition

There are various functions you can use with SQL. Things like:

- MAX and MIN
  - Suppose you want to find the maximum and minimum weight of a football player in the table above.
  - SELECT MAX(weight), MIN(weight)
    FROM players
- AVG and SUM
  - You can find the average or sum of a given column. For example:
  - SELECT AVG(weight), SUM(weight)
    FROM players
- COUNT
  - Count will return the count of the number of rows returned.
  - Suppose I wanted to count the number of quarterbacks on the team.
    - SELECT COUNT(*)
      FROM players
      WHERE position = “QB”
    - COUNT(*) was used, but COUNT(playerID) would have also worked. COUNT will only count rows that do not have NULL values, so it is a good idea to count on the key or another field that cannot be NULL
- LIKE
  - Using the LIKE keyword allows us to place further conditions onto strings.
  - Suppose I wanted to return the information for any player that has the last name of Brown.
    - SELECT *
      FROM players
      WHERE name LIKE “%Brown”
  - The % acts as a “wildcard” and represents any amount of characters in its place. If I wanted to allow for “Brown” to be found anywhere in the string, I would use LIKE “%Brown%”
- GROUP BY
  - GROUP BY is an operator that can be placed at the end of a SQL query. It allows you to “group” rows together based on a certain criteria.
  - Suppose I wanted to find the average weight for each football position:
    - SELECT position, AVG(weight)
      FROM players
      GROUP BY position
- ORDER BY

- The ORDER BY keyword is used to sort on a specific column.
- The default is to sort in ascending (ASC) order, but you can also specify to sort descending (DESC)
- Suppose I wanted to return the player information in alphabetical order:
  - SELECT *
    FROM players
    ORDER BY name
- Suppose I wanted to return the player information in the order of heaviest player first:
  - SELECT *
    FROM players
    ORDER BY weight DESC

Using a database with python

In CS 2316, we use the pymysql module to connect to SQL databases.

To connect to a database, we use the following command:

```
db = pymysql.connect(host = ‘hostname.com’, passwd = 'password', user = ‘username’,db=’databaseName’)
```

By doing this, a reference to the database is now stored in db, which we need to execute queries.

To execute a query, we need to use a cursor.

First, you need to initiate a cursor:

```
myCursor = db.cursor()
```

Let’s say I want to insert a new player into my “players” table.

```
myQuery = “INSERT INTO players VALUES (%s, %s, %s, %s)”
```

- The %s are acting as placeholders for values that I will give the query. Instead of hard-coding the player’s information, I will allow myself flexibility to insert multiple players with the same query.

Now, I need to execute the query:

```
myCursor.execute(myQuery, (“42”, “John Grisham”, “QB”,  “110”))
```

The second parameter of myCursor.execute is a tuple of what I want to insert into my placeholders (%s).

Since I just inserted new information into the database, it would be a good idea to commit my changes. You should do this every time you finish your work with the database:

```
db.commit()
```

Let's say that instead, I wanted to get information from the database – all the players' information.

From before, we know that the query to use is:

```
query = "SELECT * FROM players"
myCursor.execute(query)
```

Now, the information I want is stored within myCursor, but how do I get it out?
There are a few methods:

`firstRow = myCursor.fetchone()`

- Now, a tuple containing the first row of information gathered is stored in firstRow.
- firstRow = (21, "Joe Jackson", "QB", 220)

`allRows = myCursor.fetchall()`

- Now, a tuple containing a tuple representing each row is stored in allRows
- allRows = ((21, "Joe Jackson", "QB", 220), (13, "George Burdell", "DE", 270), (1, "Ron Howard", "DT", 350))

Use a for loop to add entries to a list:

```
myList = []
for entry in myCursor:
      myList.append(entry)
```

- This method would generally be useful if you wanted to do some sort of conditional check in your for loop as you are processing the data.

After you are finished using your cursor, remember to close it!
Do this by:

```
myCursor.close()
```

For practice:

Write a function, **giveMeInfo**, which will connect to a database, retrieve all the rows in a table, and return the information in a list of lists. You can make up the database information, but use the table name **basicInfo**.

Write a function, **insertNewPlayers**, which will insert new player information into our **players** table. The function will take in a parameter, **playerInfo**, which is a list of lists of the player information, and a parameter, **db**, which is the database object. The list could be of any length. You should insert all the information into the table.
Ex playerInfo list: [[43, “Bob Bradley”, “RB”, 210], [91, “Cool Kid”, “TE”, 250]]

XML

The XML module in python should be imported as follows:
import xml.eTree.elementTree as et
Importing a module “as” another name allows for quicker access.

You can read about XML fundamentals in the course notes:
http://www.summet.com/dmsi/html/xml.html

Here is an example function that creates a XML tree.

```
import xml.etree.ElementTree as et

def myXML():
      Grandmother = et.Element("Grandma", age="90", name="Kathleen")
      myTree = et.ElementTree(Grandmother)
      Mother = et.SubElement(Grandmother, "Mom", age="50", name="Ruth")
      Mother.text = "Clean your room!"
      Aunt = et.Element("Aunt", num="54", letter="Cathy")
      Grandmother.append(Aunt)
      You = et.SubElement(Mother, "You", age="20", name="Bob")
      You.text = "I hate cleaning my room!"
```

Note that a subelement can be added to an XML tree in two ways:

1. You can create the element separately and later append it to its parent.
   ```
   myEl = et.Element(“thisName”)
   root.append(myEl)
   ```
2. You can create it as a subelement, for which you specify its parent as the first parameter:
   ```
   myEl = et.SubElement(root, “thisName”)
   ```

Here are some things you can do with an XML tree:

Let’s say I wanted to get some information about “Mother”
`Mother.attrib` -> returns a dictionary of the attributes. In this case,
`{“age”: “50”, “name”: “Ruth”}`
`Mother.text` -> returns the text contained in the xml tag

```
      “Clean your room!”
```

If I wanted to write the XML tree out to a file, I would use
```
myTree.write(“filename.xml”, “UTF-8”)
“UTF-8” is the encoding type of the file.
```

Alternatively, to open an xml file, I would use:
```
myTree = et.parse(“filename.xml”)
```
Then, to find the root of myTree:
```
myRoot = myTree.getroot()
```

What if I want to find a certain branch of the tree?
```
foundBranch = myRoot.find(“Branch Name”)
```

For practice, draw our what this tree would look like.

Practice Problems (Possible) Solutions:

```
import pymysql

def giveMeInfo():

    db = pymysql.connect(host = 'myhost.edu', passwd = 'password', user =
      'cs2316', db='cs2316db')
    query = "SELECT * FROM basicInfo"
    myCursor = db.cursor()
    myCursor.execute(query)
    myList = [] # Create big list to return
    for row in myCursor: # Iterate through the cursor
        rowList = [] # For every row, start a new list
        for item in row: # Iterate through each item in the row
            rowList.append(item) # Add each item to the row list
        myList.append(rowList) # Append the row list to the new list
    myCursor.close()
    db.commit()
    return rowList


def insertNewPlayers(playerInfo, db):

    myCursor = db.cursor()
    query = "INSERT INTO players VALUES(%s, %s, %s, %s)"
```

```
for player in playerInfo:
    myCursor.execute(query, (str(player[0]), str(player[1]),
        str(player[2]), str(player[3])))

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