CS3 Database Systems

Assignment 2. Due 4pm Monday, 8 November, 2010

Submission instructions. Prepare a folder with the following files:

- An answers file in pdf format – call it `answers.pdf`
- For each program you write, a file a file containing that program.
- For each program you write, a file containing the output of that program.
- Files containing any input data you used.

All data and program files should be references in `answers.pdf`

Submit hour homework no later than *4pm on Monday 8 November*

To submit your homework, use `submit cs3 dbs 2` *myfolder*, where *myfolder* is the name of your folder.

In doing the SQL homework, please (a) write your queries as clearly as possible (use intermediate tables or views) (b) comment your queries, and (c) make sure your programs can be easily run and checked.

Please note that, when you starting to work with databases, it is a very bad idea to create a database by doing manual inserts. Instead, write scripts to do this and keep the scripts. That way, if you have some kind of disaster you can easily rebuild the database.

1. [40 points](Scottish History) The SQL script on the course web site is a partial and slightly doctored list of Scottish monarchs. It consists of a single table of the following form:

name	start	stop
Aed	877	878
Alexander I	1107	1124
Alexander II	1214	1249
Alexander III	1249	1286
Constantine I	862	877
Constantine II	900	943
Constantine III	995	997
Culen	966	971
...	...	...

Use SQL to answer the following queries. For complex queries define intermediate tables (views) if needed and describe what those views represent. Hand in both the queries and the output they generate.

(a) A list of all the monarchs who reigned before the year 1000.

Answer:

```
SELECT name
FROM monarchs
WHERE start < 1000;
```

(b) A list which gives pairs of monarchs consisting of a monarch and their immediate successor – if one exists. Note: a immediately succeeds b if a reigned after b and no other monarch reigned between a and b.

Answer:

```
SELECT m1.name, m2.name
FROM monarchs m1, monarchs m2
WHERE m1.start = m2.start AND
NOT EXISTS ( SELECT * FROM monarchs m3
             WHERE m1.start < m3.start AND
             m3.start < m2.start);
```

Note: the answer to this and some other questions would be complicated if there were monarchs who reigned for less than a year. These answers assume there isn't, but maybe bonus points should be given to people who attempt to deal with this situation.

(c) The name(s) of the monarch(s) with the longest reign(s).
Answer:

```
SELECT m1.name, m1.stop - m1.start AS duration
FROM monarchs m1
WHERE m1.stop - m1.start >= ALL (SELECT stop - start FROM monarchs);
```

Note: duration is not required.

(d) The average duration of the reigns of all monarchs.
Answer:

```
SELECT AVG(stop - start) AS duration FROM monarchs;
```

(e) A list of the interregna. An interregnum is a period between two monarchs for which there was no monarch. Give the start and stop dates.
Answer:

```
SELECT m1.stop as start, m2.start as stop
FROM monarchs m1, monarchs m2
WHERE m1.stop < m2.start
AND NOT EXISTS ( SELECT * FROM monarchs m3
                 WHERE m1.stop ¡= m3.start AND
                 m3.start < m2.start);
```

Notes: "/" in SQL (or at least psql) does integer division. ORDER BY is not needed, but nice

(f) A list of the centuries in which there was a reigning monarch together with the average duration of the reigns of those monarchs whose reigns *ended* in that century. (Use the starting year, e.g. 1100, to describe the centuries.)
Answer:

```
SELECT 100 *(stop / 100) as century, AVG(stop-start)
FROM monarchs
GROUP BY stop / 100
ORDER BY century;
```

Notes: "/" in SQL (or at least psql) does integer division. ORDER BY is not needed, but nice

2. [60 points] An interesting way of representing XML in a relational database is to record, for each node, the start position and the end position. Consider, for example, the XML document

$$\underset{1}{\underline{\langle\text{db}\rangle}}\ \underset{2}{\underline{\langle\text{person}\rangle}}\ \underset{3}{\underline{\langle\text{name}\rangle}}\ \underset{4}{\underline{\text{joe}}}\ \underset{5}{\underline{\langle/\text{name}\rangle}}\ \underset{6}{\underline{\langle\text{tel}\rangle}}\ \underset{7}{\underline{\text{1234}}}\ \underset{8}{\underline{\langle/\text{tel}\rangle}}\ \underset{9}{\underline{\langle/\text{person}\rangle}}\ \underset{10}{\underline{\langle\text{cat}\rangle}}\ \underset{11}{\underline{\langle\text{name}\rangle}}\ \underset{12}{\underline{\text{kitty}}}\ \underset{13}{\underline{\langle/\text{name}\rangle}}\ \underset{14}{\underline{\langle/\text{cat}\rangle}}\ \underset{15}{\underline{\langle/\text{db}\rangle}}$$

Here each tag and each text string has been annotated by its position in the document.

Elements =

start	stop	tag
10	14	cat
3	5	name
1	13	db
2	9	person
6	8	tel
11	13	name

TextNodes =

pos	pcdata
12	kitty
4	joe
7	1234

The XML document can be described by two tables: one for elements which has the positions of its start and stop tags and its tag name; and one for character data that has the positions of the text nodes. Note that the start position is a key for the elements table and the position is a key for the text nodes table.

In this question we'll only consider XML with text and element nodes (no attributes). Moreover we'll assume that there is no mixed content.

(a) Write a program in your favourite programming language that uses a SAX parser to construct the database. On the course Web site there is an Python example of using a sax parser to traverse an XML document to obtain the relevant information. There is also information on how to call SQL from Python. *In this and subsequent questions your program should not require main memory storage of the document. All your programs should be capable of working on huge documents by building a large database.*

Answer: Here is my code. I made slight modifications to the SAX example given out. The function `createtables` creates the tables, and populates them by calling the SAX parser. Sample output in exhibit 2a at end of this answer sheet.

```
import psycopg2
import xml.sax

# This is a modification of the code given out to use SAX to populate
# an e(lement) table and a c(haracter) table over a cursor, cur, that
# are passed in as parameters to the constructor
class TableFiller(xml.sax.handler.ContentHandler):
    def __init__(self,cursor,etable,ctable):
        self.nodecount = 0  #The serial number of the tag or text node
        self.buffer = ""    #For collecting character strings
        self.stack = []     #The stack "remembers" the matching start tag.
        self.cursor = cursor
        self.etable = etable
        self.ctable = ctable

    def startElement(self,nme,att):
        self.stack.append((self.nodecount,nme))
        self.nodecount=self.nodecount+1

    def endElement(self,nme):
        (tcount,tnme)=self.stack.pop()
        if tnme <> nme: print "non-well-formed document"
        ss = self.buffer.strip()
        if len(ss) <> 0:
            cur.execute("INSERT INTO "+ self.ctable + " (pos, pcdata) " +
                        "VALUES(%s,%s);",(self.nodecount,ss))
            self.nodecount=self.nodecount +1
            self.buffer=""
        cur.execute("INSERT INTO "+ self.etable + " (start, stop, tag) " +
                    "VALUES(%s,%s,%s);", (tcount, self.nodecount, nme))
        self.nodecount=self.nodecount+1
    def characters(self,data):
        self.buffer = self.buffer+data


# This function reads an XML file and creates the named tables
# using the cursor c.

def createtables(c,xmlfile, etable, ctable):
    #create the tables
    c.execute("CREATE TABLE " + etable + " (start INT, stop INT, tag TEXT);")
    c.execute("CREATE TABLE " + ctable + " (pos INT, pcdata TEXT);" )
    #make a parser and handler
    parser=xml.sax.make_parser()
    parser.setContentHandler(TableFiller(c,etable,ctable))
    # and run it over the file
    parser.parse(xmlfile)

# Now test it
conn = psycopg2.connect("dbname=peter")
```

```
cur = conn.cursor()
createtables(cur,"emps.xml","Elements", "TextNodes")
conn.commit()
cur.close()
conn.close()
```

(b) Write a program – something like descendants(c, t) where c is a positions and t is a tag name that generates an SQL query to generate the result of the XPATH expression .//t applied at the context node c. All the computation should be done in SQL.

Answer:

```
import psycopg2

# I've parametrised my getdescendants function with the cursor,
# the name of the element table, the context node and the tag name.
# The function returns a python list of the positions of the XPATH result
# If there were ever a possibility that the XPATH result would be huge, it
# should be changed to return a Python generator.



def getdescendants(c, etable, context, tag):
    # Set up the query.  We do this in two stages: we first use Python string
    # manipulation to "plug in" in the relevant table names.  Then we use
    # psycopg2 conventions (%%s) for adding in the remaining fields.
    query ="""
SELECT e2.start
FROM %s as e1, %s as e2
WHERE e2.tag =  %%s AND e1.start = %%s
AND e1.start < e2. start and e2.stop < e1.stop;
"""%(etable,etable,)
    # print query # in case you want to see it
    c.execute(query,(tag,context))
    result = []
    while True:
        t = c.fetchone()
        if t == None: break
        result.append(t[0])
    return result

#Now test the whole thing with .//tel applied at context node 1

conn = psycopg2.connect("dbname=peter")
cur = conn.cursor()
print getdescendants(cur,"Elements", 1, "tel")
cur.close()
conn.close()
```

This produced the output `[5, 12, 26]`

(c) Write a program – something like children(c, t) where c is a position and t is a tag name that generates an SQL query to generate the result of the XPATH expression $./t$ applied at the context node c. (Note: this is harder than the previous question)

Answer:

```
import psycopg2
# To make this understandable, here is the "generic" SQL query that returns
# the children with tag TAG of the context node CONTEXT defined by its start
# position in table ELEMENTS.  All we do is find the elements that are "inside"
# the context node, but are not inside an element that is inside the context
# node
#
# SELECT e2.start
# FROM ELEMENTS as e1, ELEMENTS as e2
# WHERE e2.tag =  TAG AND e1.start = CONTEXT
# AND e1.start < e2. start and e2.stop < e1.stop
# AND NOT EXISTS (SELECT * FROM ELEMENTS as e3
#                 WHERE e1.start < e3.start AND e3.start < e2.START
#                 AND e2.stop <e3.stop and e3.stop <e 1.stop);

# Parameterisation works as in the previous question

def getchildren(c, etable, context, tag):
    # Set up the query as before
    query ="""
SELECT e2.start
FROM %s as e1, %s as e2
WHERE e2.tag =  %%s AND e1.start = %%s
AND e1.start < e2. start and e2.stop < e1.stop
AND NOT EXISTS (SELECT * FROM %s as e3
                WHERE e1.start < e3.start AND e3.start < e2.START
                AND e2.stop <e3.stop and e3.stop <e1.stop);
"""%(etable,etable,etable)
    # print query # in case you want to see it
    c.execute(query,(tag,context))
    result = []
    while True:
        t = c.fetchone()
        if t == None: break
        result.append(t[0])
    return result

#Now test the whole thing with .//tel applied at context node 1

conn = psycopg2.connect("dbname=peter")
cur = conn.cursor()
print getchildren(cur,"Elements", 1, "tel")
cur.close()
conn.close()
```

This produced the result `[5]`

(d) Write an SQL query to check whether a pair of such tables is consistent with some XML document. I.e., the tables are "well-formed". You should assume that the node positions are ordered but it is not necessary to assume that adjacent positions are consecutively numbered. E.g. $\frac{\langle\text{cat}\rangle}{5}\frac{\langle\text{name}\rangle}{11}\frac{\text{kitty}}{12}\frac{\langle/\text{name}\rangle}{15}\frac{\langle/\text{cat}\rangle}{33}$ is an acceptable numbering. The SQL query should return a table containing 1 if the tables are consistent with XML and 0 otherwise.

Answer: To do this we need to check that the elements table is properly nested – there are no tuples t1,t2 such that t1[start] ¡ t2[start] and t1[stop] , t2[stop]. We also want to check that there are no text nodes whose pos is between adjacent start nodes or between adjacent stop nodes.

The requirement of this question – to produce a SQL table containing 1 for well-formed and 0 otherwise is annoying (sorry about that!) Obviously what is wanted is a return in Python (or the host language) of True or False.

Here is a SQL query that produces a non-empty result if the ELEMENTS table fails to obey the nested tags rule:

```
SELECT DISTINCT 0 AS flag
FROM ELEMENTS e1, ELEMENTS e2
WHERE e1.start < e2.start AND e2.stop < e2.stop
```

And here is a query on the TEXTNODES and ELEMENTS tables to check that every text node is enclosed in an element and that element (no mixed content) does not contain any other element

```
SELECT DISTINCT 0 AS flag
FROM TEXTNODES t1
WHERE NOT EXISTS (SELECT * FROM ELEMENTS e1 //There must be a surrounding element
                  WHERE e1.start < t1.pos AND t1.pos < e1.stop
                  AND NOT EXISTS ( SELECT * FROM ELEMENTS e2
                  WHERE e1.start < e2.start and e2.stop < e1.stop))
```

Similarly for stop tags. Finally (and apologies again for this needless step), suppose T is a table with a flag column that contains either a 1 or is empty. Postgres allows one to construct a table with a single 1 by `SELECT 1 AS flag;` So to get the desired answer: `SELECT MIN flag FROM (SELECT 1 AS flag UNION T)`

The code is shown in exhibit 2 with tests.

(e) Write a function – something like delete(s) to delete a set of nodes from the database. The set s is a set of positions, and the function should have the effect of deleting the node at each position in s from the document tree together will all its descendants. To do this construct a temporary table containing the positions in s.

Answer: Suppose the nodes to be deleted are in single column (id) table deletetable. Then the following code will delete those nodes (if they exist) and any node contained in them.

```
– Deletes any text node contained in an element whose position is in deletetable
DELETE FROM TEXTNODES t WHERE t.pos IN
( SELECT t1.pos FROM TEXTNODES t1, deletetable d, ELEMENTS e
  WHERE t1.pos = d.id OR (e.start = d.id AND e.start < t1.pos AND t1.pos < e.stop));

– Deletes any element contained in an element whose position is in deletetable
DELETE FROM ELEMENTS e where e.start IN
( SELECT e2.start FROM deletetable d, ELEMENTS e1, ELEMENTS e2
  WHERE e1.start = d.id AND e2.start >= e1.start AND e2.stop <= e1.stop);
```

The only issue is creating the the table of nodes to be deleted. It is best to do it in one go rather than repeatedly calling an SQL query for each member of a python list as this is *very* expensive.

Exhibit 2e shows the code for this and sample output.

(f) Write a function – something like generateXML(e, t, f) that generates an XML document from a well-formed database consisting of an elements table e and a text nodes table t and writes it into a file f. The function should be capable of working on very large databases. Hint: use SQL's ORDER-BY. Demonstrate your function by reading in some XML, deleting a set of nodes with a given tag name, and writing the XML back to a file.

Answer: Both the SQL code and the Python code are simple. Here is the SQL code:

```
( SELECT start AS id, 'start' AS type, tag AS value FROM Elements
  UNION
  SELECT stop AS id, 'stop' AS type, tag AS value FROM Elements
  UNION
  SELECT pos AS id, 'text' AS type, pcdata AS valueb FROM TextNodes)
ORDER by id;
```

The output of such a query looks like:

```
 id  | type  |         value
-----+-------+-----------------------
   0 | start | db
   1 | start | department
   2 | start | dname
   3 | text  | manufacturing
   4 | stop  | dname
   5 | start | tel
   6 | text  | 1432
   7 | stop  | tel
...
```

The Python code uses a cursor to scan the output and write the appropriate tags and text to a file. Note that this program can work on very large XML representations. All the space management is in the database, not in the Python code.

The code to read in some XML (question 2a), delete some nodes (question 2e) and write the XML back to a file is shown in exhibit 2f.

Exhibit 2a

```
select * from Elements;
 start | stop |    tag
-------+------+------------
     2 |    4 | dname
     5 |    7 | tel
     9 |   11 | name
    12 |   14 | tel
    15 |   17 | sal
    18 |   20 | project
     8 |   21 | employee
    23 |   25 | name
    26 |   28 | tel
    29 |   31 | sal
    32 |   34 | project
    35 |   37 | project
    22 |   38 | employee
    40 |   42 | name
    43 |   45 | sal
    46 |   48 | project
    39 |   49 | employee
     1 |   50 | department
...

select * from TextNodes;

 pos |         pcdata
-----+------------------------
   3 | manufacturing
   6 | 1432
  10 | Jane Dee
  13 | 6734
  16 | 50
  19 | Methods and Standards
  24 | Mary Smith
  27 | 1432
  30 | 45
  33 | Data Mining
  36 | Systems Development
  41 | John Brown
  44 | 25
  47 | Logistics
  53 | sales
  56 | 3221
  60 | Fred Beans
  63 | 3221
  66 | 32
...
```

Exhibit 2d

The code:

```
import psycopg2


def wellformed(c, etable, ttable):  #cursor, elements and text nodes
    query ="""
SELECT MIN(TEMP.flag) FROM
( SELECT 1 as flag

  UNION

  SELECT DISTINCT 0 AS flag
  FROM %s e1, %s e2
  WHERE e1.start < e2.start AND e2.stop < e2.stop

  UNION

  SELECT  DISTINCT 0 AS flag
  FROM %s t1
  WHERE NOT EXISTS (SELECT * FROM %s e1
    WHERE e1.start < t1.pos AND t1.pos < e1.stop
    AND NOT EXISTS (SELECT * FROM %s e2
    WHERE e1.start < e2.start and e2.stop < e1.stop))
) AS TEMP;
"""%(etable,etable,ttable,etable,etable)
    # print query # in case you want to see it
    c.execute(query)
    result = []
    t = c.fetchall()
    print t
    return t[0][0]== 1
```

Exibit 2d – Some tests

```
# The following tests show the contents of the result table and the (more sensible)
# python return value
conn = psycopg2.connect("dbname=peter")
cur = conn.cursor()

print "Test 1 -- on the employees/departments data given with the homework."
print wellformed(cur, "Elements", "TextNodes")
print

print "Create empty element and text node test tables"
cur.execute("""
CREATE TABLE els(start int, stop int, tag text);
CREATE TABLE txts (pos int, pcdata text);""")
print

print "Test 2 -- on empty element and text tables -- should be well-formed"
print wellformed(cur,"els","txts")
print

print "Test 3 -- insert a text node (5, ’Some text’) (no surrounding element)"
cur.execute("INSERT INTO txts VALUES(%s,%s)",(5,"Some text"))
print wellformed(cur,"els","txts")
print

print "Test 4 -- now insert a surrounding element (0, 10,’tag1’) "
cur.execute("INSERT INTO els VALUES(%s,%s,%s)",(0, 10,"tag1"))
print wellformed(cur,"els","txts")
print

print "Test 6 -- now insert an element (1, 3, ’tag2’) that creates mixed content"
cur.execute("INSERT INTO els VALUES(%s,%s,%s)",(1, 3,"tag2"))
print wellformed(cur,"els","txts")
print

cur.close()
conn.close()
```

Exhibit 2d – output from these tests

```
>>> Test 1 -- on the employees/departments data given with the homework.
[(1,)]
True

Create empty element and text node test tables

Test 2 -- on empty element and text tables -- should be well-formed
[(1,)]
True
```

```
Test 3 -- insert a text node (5, 'Some text') (no surrounding element)
[(0,)]
False

Test 4 -- now insert a surrounding element (0, 10,'tag1')
[(1,)]
True

Test 6 -- now insert an element (1, 3, 'tag2') that creates mixed content
[(0,)]
False
```

Exhibit 2e – code

```
import psycopg2


#l is a python list of nodes to be deleted.
def deletenodes(c, etable, ttable, l):
    #First construct the SQL command to insert l into deletetable
    if l == []: delstring = ""
    else:
        delstring = "(" + str(l[0]) + ")"
        for x in l[1:]: delstring = delstring + ", (" + str(x) + ")"
    # Set up the query as before
    query ="""
CREATE TABLE deletetable(id int);
INSERT INTO deletetable VALUES %s;  -- Python should construct this in one shot


-- Deletes any text node contained in an element whose position is in deletetable
DELETE FROM %s t WHERE t.pos IN
 (SELECT t1.pos FROM %s t1, deletetable d, %s e
   WHERE t1.pos = d.id OR (e.start = d.id AND e.start < t1.pos AND t1.pos < e.stop));

-- Deletes any element contained in an element whose position is in deletetable
DELETE FROM %s e where e.start IN
 (SELECT e2.start FROM deletetable d, %s e1, %s e2
  WHERE e1.start = d.id AND e2.start >= e1.start AND e2.stop <= e1.stop);

DROP TABLE deletetable;
"""%(delstring,ttable,ttable,etable,etable,etable,etable)
    # print query # in case you want to see it
    c.execute(query)

# Test

conn = psycopg2.connect("dbname=peter")
cur = conn.cursor()
deletenodes(cur, "Elements", "TextNodes", [51, 5, 150, 108])
conn.commit()
cur.close()
conn.close()
```

Exhibit 2e – results from class data after deletion

```
peter=> select * from elements;
 start | stop |    tag
-------+------+------------
     2 |    4 | dname
     9 |   11 | name
    12 |   14 | tel
    15 |   17 | sal
    18 |   20 | project
     8 |   21 | employee
    23 |   25 | name
    26 |   28 | tel
    29 |   31 | sal
    32 |   34 | project
    35 |   37 | project
    22 |   38 | employee
    40 |   42 | name
    43 |   45 | sal
    46 |   48 | project
    39 |   49 | employee
     1 |   50 | department
    88 |   90 | dname
    91 |   93 | tel
    95 |   97 | name
    98 |  100 | tel
   101 |  103 | tel
   104 |  106 | sal
   107 |  109 | project
   110 |  112 | project
    94 |  113 | employee
   115 |  117 | name
   118 |  120 | tel
   121 |  123 | sal
   124 |  126 | project
   114 |  127 | employee
    87 |  128 | department
     0 |  129 | db
(33 rows)
```

Exhibit 2f – the program

```
def genXML(c, file, etable, ttable):
    # Set up the query as before
    query ="""
(SELECT start AS id, ’start’ AS type, tag AS  value FROM %s
 UNION
 select stop AS id, ’stop’ AS type, tag AS value FROM %s
 UNION
 select pos AS id, ’text’ AS type, pcdata AS valueb FROM %s)
ORDER by id;
"""%(etable,etable,ttable)
    # print query # in case you want to see it
    outs = open(file, ’w’)
    c.execute(query)
    while True: #Now iterate through the result
        s = c.fetchone()
        if s == None: break
        (id,type,val) = s
        if type == "start": outs.write("<"+val+">")
        elif type == "stop":  outs.write("</"+val+">")
        else: outs.write(" " + val + " " )
    outs.close()

conn = psycopg2.connect("dbname=peter")
cur = conn.cursor()
createtables(cur,"emps.xml","Elements", "TextNodes")    #question 2a
deletenodes(cur, "Elements", "TextNodes", [51, 5, 150, 108]) #question 2e

genXML(cur,"test.xml","Elements", "TextNodes")
cur.close()
conn.close()
```

Exhibit 2e – the generated XML file

(I passed the output file through xmllint to format it nicely)

```
<db>
  <department>
    <dname> manufacturing </dname>
    <employee>
      <name> Jane Dee </name>
      <tel> 6734 </tel>
      <sal> 50 </sal>
      <project> Methods and Standards </project>
    </employee>
    <employee>
      <name> Mary Smith </name>
      <tel> 1432 </tel>
      <sal> 45 </sal>
      <project> Data Mining </project>
      <project> Systems Development </project>
    </employee>
    <employee>
      <name> John Brown </name>
      <sal> 25 </sal>
      <project> Logistics </project>
    </employee>
  </department>
  <department>
    <dname> research </dname>
    <tel> 7776 </tel>
    <employee>
      <name> Sara Lee </name>
      <tel> 5554 </tel>
      <tel> 3221 </tel>
      <sal> 32 </sal>
      <project/>
      <project> Data Mining </project>
    </employee>
    <employee>
      <name> Jim Bean </name>
      <tel> 1223 </tel>
      <sal> 25 </sal>
      <project> Methods and Standards </project>
    </employee>
  </department>
</db>
```