Problem Set 3

Preliminaries

In your work on this assignment, make sure to abide by the collaboration policies of the course. ***All of the problems in this assignment are individual-only problems that you must complete on your own.***

If you have questions, please come to office hours, post them on Piazza, or email `cs460-staff@cs.bu.edu`.

Make sure to submit your work on Gradescope, following the procedures found at the end of Part I and Part II.

Part I

due by 11:59 p.m. on Wednesday, March 20, 2019
50 points total

Creating the necessary file

This part of the assignment will all be completed in a single PDF file. To create it, you should do the following:

1. Open the template that we have created for Part I in Google Docs: ps3_partI
2. Select *File->Make a copy...*, and save the copy to your Google Drive using the name `ps3_partI`.
3. Add your work for all of the problems from Part I to this file.
4. Once you have completed Part I, choose *File->Download as->PDF document*, and save the PDF file on your machine. The resulting PDF file (`ps3_partI.pdf`) is the one that you will submit. See the submission guidelines at the end of Part I.

Problem 1: Conflict serializability

10 points total

Below are two schedules in which actions by three transactions (T1, T2, and T3) are interleaved.

schedule 1:

```
r3(A); w2(B); r1(B); w1(C); w2(A); r3(C)
```

schedule 2:

```
w3(A); r1(A); r1(B); r2(C); w3(C); r2(B)
```

For each of these schedules, you should take the following steps:

1. In your copy of the `ps3_partI` template (see above), edit the diagram that we have provided for the schedule, making whatever changes are needed to construct the schedule's precedence graph.
2. State whether the schedule is conflict serializable.
3. If the schedule *is* conflict serializable, state one possible equivalent serial schedule. If the schedule is *not* conflict serializable, explain briefly why the schedule is not equivalent to the

serial schedule T1; T2; T3. Your explanation should include a discussion of why particular actions from the original schedule are not consistent with that serial schedule.

Problem 2: Two-phase locking and isolation

10 points total; 2 points each part

Consider these two transactions:

```
T1: r(A); w(A); r(B); c
T2: r(B); w(B); r(A); c
```

1. The following is the beginning of one possible schedule of these transactions, with appropriate lock instructions added:

T1	T2
	xl(B)
	r(B)
	w(B)
sl(A)	
r(A)	
	sl(A)
	u(B)
...	...

We have included this partial schedule in the `ps3_partI` template.

Complete this schedule (i.e., provide the remainder of the actions) in a way that follows the two-phase locking rule, as well as the rules for which operations are allowed when holding shared and exclusive locks. The completed schedule should allow both transactions to finish without being rolled back and restarted.

Notes:

- When completing the schedule table provided in the template, please put only one action per “line” of the table so that the order in which the actions are completed will be clear.

- You should assume that shared locks *can* be upgraded, and that there are no update locks.
- Don't forget that the order of the items within a given transaction cannot change. For example, regardless of how you interleave the actions from T1 and T2, T1's write of A must still come before T1's read of B, because that is the order in which those actions occur when T1 is executed on its own.

2. The partial schedule that we gave you for part 1 does not observe strict locking. Explain briefly what aspect(s) of the schedule prevent it from being strict.
3. Is your completed schedule recoverable? Explain briefly why or why not.
4. If your original completed schedule *is* recoverable, construct an alternative schedule for these two transactions that is *not* recoverable. If your original completed schedule is *not* recoverable, construct an alternative schedule that *is* recoverable. Give the schedule in table form as shown above. The new schedule should still be non-serial (see the note below), and it should allow both transactions to finish without being aborted and restarted.

   Notes:

   - ***Important:*** In order for a schedule to be non-serial, it must be the case that there is at least some interleaving of the *reads and/or writes* performed by the transactions. It is *not* enough to simply delay one or more of the transaction commits (e.g., all of T1 except its commit, followed by all of T2, followed by T1's commit).
   - The new schedule may begin with the partial schedule given at the start of the problem, but doing so is not required.
   - You should again assume that shared locks can be upgraded, and that there are no update locks.

5. Consider the recoverable schedule that you gave for either question 1 or question 4. Is that schedule cascadeless? Explain briefly why or why not.

Problem 3: Lock modes

9 points total; 1.5 points each part

Consider the following partial schedule of transactions T2, T3, and T4:

```
ul3(A); r3(A); xl2(C); w2(C); sl4(B); r4(B); ...
```

Given this partial schedule, which of the following lock requests would be granted if it were the **next operation** in the schedule, and which would be denied?

1. `xl2(B)`
2. `sl3(B)`

3. `sl4(A)`
4. `ul2(A)`
5. `ul3(B)`
6. `ul4(C)`

Fill in the provided table with the system's response to each request, along with a brief explanation of your answer. (Remember: `uli(X)` indicates Ti's request of the update lock for X.)

Problem 4: Deadlock detection

9 points total

For each sequence of operations below, determine whether deadlock will occur under *rigorous* two-phase locking. You should assume that:

- an exclusive lock for an item is requested just before writing that item
- a shared lock for an item is requested (if needed) just before reading that item
- a transaction commits immediately after completing its final read or write
- upgrades of shared locks are allowed.

If deadlock occurs, show the partial schedule in table form (including lock operations and any commits) up to the point of deadlock, along with the waits-for graph at the point of deadlock.

If deadlock does not occur, show the full schedule, including lock operations and commits. In that case, you do **not** need to include the waits-for graph.

In either case, your schedule should include any changes to the sequence of operations that occur because a transaction is forced to wait for a lock. If a transaction waits for a lock that is subsequently granted, you should include two lock requests – one for the initial request, and one at the point at which the lock is granted. If two transactions wait for a lock held by the same other transaction, and that other transaction subsequently commits, you should assume that the waiting transaction with the smaller transaction number is granted its request first.

> **Important**
>
> Don't forget that if a transaction is forced to wait for a lock, it *cannot* make *any* forward progress until that lock is granted.

sequence 1:

```
r1(A); w1(B); r2(B); w2(C); r3(C); w3(A); w1(A)
```

sequence 2:

```
w3(A); r1(A); w1(B); r2(B); w2(C); r3(C); w2(A)
```

Problem 5: Timestamps and multiple versions

12 points total; 4 points each part

Consider the following sequence of operations:

```
s1; s2; s3; s4; s5; r2(A); w4(A); r3(A); w1(A); w2(A); r5(A); c1; c2; c3; c4
```

where `si` indicates the start of transaction `Ti`, which is when its timestamp is assigned, and `ci` indicates the commit of transaction `Ti`.

Assume that the transactions are assigned the following timestamps, based on the order in which they start:

- TS(T1) = 10
- TS(T2) = 20
- TS(T3) = 30
- TS(T4) = 40
- TS(T5) = 50

and that RTS(A) and WTS(A) are both initially 0.

Given these assumptions:

1. In the first table for this problem in `ps3_partI`, we've filled in the rows for the first two operations. Complete the table to show how the system would respond to the remaining read and write requests when it is using **regular timestamp-based concurrency control *without* commit bits**.

    a. In the first column of the table, put the requested operation.

    b. In the second column of the table, indicate:

    - whether the specified operation is allowed, denied, or ignored
    - any action taken against the requesting transaction (e.g., "roll back" or "make wait").

    c. In the third column of the table, you should do one of the following:

    - If the action is allowed, summarize any changes to the state maintained for item A, as we've done in the first two rows.
    - If the action is denied, include a brief explanation. For example, if a transaction T7 tried to read item B and its read were too late, you would include something like

this:

```
TS(T7) < WTS(B)
```

d. You do ***not*** need to restart a transaction that is rolled back, which means that you can skip any requests by a transaction that come after the point at which it is rolled back.

e. Because commits don't have an effect when we're not using commit bits, you do ***not*** need to include the commit actions (`c1`, `c2`, etc.) in this table.

2. Complete the second table that we've provided to show the system's response to this sequence of operations if the DBMS is using **regular timestamp-based concurrency control *with* commit bits**.

   a. Notes a-d from 5.1 also apply here.

   b. In addition to the reads and writes, you should include the commits at the end of the sequence since they may cause a change in state.

   c. If a transaction is made to wait, you should include an additional row for the operation in question when the wait comes to an end.

3. *We will cover the material needed for this part of the problem on the Monday after spring break.*

   Explain what will happen in response to this sequence of operations if the DBMS uses ***multiversion* timestamp-based concurrency control *without* commit bits**.

   a. Notes a-e from 5.1 also apply here.

   b. In the third column, make sure to specify which version's state is being updated (e.g., "create A(t) with RTS = 0" or "RTS(A(t)) = 20", where t is the timestamp of the version).

   c. Similarly, if a transaction is allowed to read A, make sure to indicate in the second column which version it is allowed to read.

Submitting your work for Part I

1. Once you have completed Part I in Google Drive, choose *File->Download as->PDF document*, and save the resulting file (`ps3_partI.pdf`) on your machine.

2. Login to Gradescope by clicking the following link:

   Gradescope

3. Once you are in logged in, click on the box for ***CS 460***.

4. Click on the name ***PS 3, Part I*** in the list of assignments. You should see a pop-up window labeled ***Submit Assignment***. (If you don't see it, click the ***Submit*** or ***Resubmit*** button at the

bottom of the page.)

5. Choose the ***Submit PDF*** option, and then click the *Select PDF* button and find the `ps3_partI.pdf` that you created in step 1. Then click the *Upload PDF* button.

6. You should see an outline of the problems along with thumbnails of the pages from your uploaded PDF. For each problem in the outline:

   - Click the title of the problem.
   - Click the page(s) on which your work for that problem can be found.

   As you do so, click on the magnifying glass icon for each page and doublecheck that the pages that you see contain the work that you want us to grade.

7. Once you have assigned pages to all of the problems in the question outline, click the *Submit* button in the lower-right corner of the window.

8. You should see a box saying that your submission was successful. Click the `(x)` button to close that box.

9. You can use the ***Resubmit*** button at the bottom of the page to resubmit your work as many times as needed before the final deadline.

Important

- It is your responsibility to ensure that the correct version of a file is on Gradescope before the final deadline. ***We will not accept any file after the submission window for a given assignment has closed, so please check your submission carefully using the steps outlined above.***

- If you are unable to access Gradescope and there is enough time to do so, wait an hour or two and then try again. If you are unable to submit and it is close to the deadline, email your homework ***before the deadline*** to `cs460-staff@cs.bu.edu`

Part II

due by 11:59 p.m. on Wednesday, April 3, 2019
50 points total

Problem 6: A schema for an XML database

10 points

Consider the following `CREATE TABLE` commands for a relational database that captures information about songs and the artists who sing them:

```
CREATE TABLE Song(id CHAR(10) PRIMARY KEY, name VARCHAR(64),
    duration INTEGER, genre VARCHAR(10), best_chart_rank INTEGER);

CREATE TABLE Artist(id CHAR(7) PRIMARY KEY, name VARCHAR(128),
    label VARCHAR(30), dob DATE, primary_genre VARCHAR(10));

CREATE TABLE Sings(songID CHAR(10), artistID CHAR(7),
    PRIMARY KEY(songID, artistID),
    FOREIGN KEY songID REFERENCES Song(id),
    FOREIGN KEY artistID REFERENCES Artist(id));
```

Let's assume that we want to develop an XML version of this database. In addition to translating the relational schema above, we would ideally like to capture the following details:

- The database does not include a `best_chart_rank` value for all songs.
- The database does not include a `primary_genre` value for all artists. If the primary genre of an artist is not specified, we can assume that that the artist's primary genre is `'pop'`.
- Aside from a song's `best_chart_rank` and an artist's `primary_genre`, all of the other song and artist attributes are always included.
- Every song is sung by at least one artist, but possibly more than one.
- Every artist has sung at least one song, but possibly more than one.

Write an XML DTD for the information in this database. Assume that the entire database will be stored in a single document, and ***use attributes to capture relationships between entities***.

Notes:

- Create the necessary file for your DTD by taking the following steps:
  - Open the template that we have created for this problem in Google Docs: ps3pr6
  - Select *File->Make a copy...*, and save the copy to your Google Drive using the name `ps3pr6`.

- Because you are writing an external DTD (i.e., one that is separate from the XML document that it is defining), you should omit the surrounding `!DOCTYPE` specification and its square brackets. Simply include the necessary `!ELEMENT` and `!ATTLIST` declarations, including one for the root element of the document.

- It may not be possible to capture all of these details mentioned above in your DTD. Do your best to capture as many of them as possible.

Problem 7: Converting the movie table to XML

20 points total

In this problem, you will write a series of methods that can be used to create an XML version of the `Movie` table from Problem Set 1. Your code will use the JDBC framework to connect to the SQLite database that you used in that assignment and to perform the SQL queries needed to extract the necessary data.

Getting started

1. Download the following file, which contains starter code for the program that will be used to convert the entire `Movie` table: `XMLforMovies.java`

   Review all of the code that we've provided before you start writing any new code.

2. Follow the instructions found in this document to download the JDBC driver for SQLite and add it to your classpath. You should also review the information found in that document for how to perform a query using JDBC.

3. Put a copy of the `movie.sqlite` file from Problem Set 1 in the same folder as `XMLforMovies.java`.

The methods you should write

> **Important**
>
> 1. We've provided the headers of the methods that you will implement. You must ***not*** change these headers in any way.
>
> 2. In the code that you write, you must limit yourself to the packages that we've imported at the top of the starter file. **You must *not* use classes from any other Java package.**

You will be implement five non-static methods of a class called `XMLforMovies`.

We've given you the constructor for this class, which:

- takes the name of a SQLite file that should contain a table called `Movie` with the schema outlined in Problem Set 1

- establishes a connection to the SQLite database and stores the resulting `Connection` object in a field called `db`.

Your methods will have access to this `Connection` object, and they should use it to perform whatever SQL queries are needed.

We've given you two examples of using JDBC to execute a query and process the results:

- the method called `idFor()`, which takes as input the name of a movie and performs a query to find and return the movie's id. This method will be useful when testing the methods that you write.
- the method called `createFile()`, which performs a query to obtain the id numbers of all of the movies in the `Movie` table, and which processes them one at a time.

Note that these methods – and many of the methods that you will implement – include a `throws` clause in their method header. This clause is needed because the code included in these methods may throw the exception(s) mentioned in the `throws` clause, and instead of catching them, we are simply declaring that they may be thrown.

Here are your tasks:

1. Implement the method called `simpleElem()` whose header we have provided. It takes the name and value of a simple XML element as inputs, and it should return a string with the XML for that element.

   Here are some example calls from the Interactions Pane in DrJava:

   ```
   > XMLforMovies xml = new XMLforMovies("movie.sqlite")
   > xml.simpleElem("rating", "PG-13")
   "<rating>PG-13</rating>"
   > xml.simpleElem("name", "Julianne Moore")
   "<name>Julianne Moore</name>"
   ```

   Note that there should be **no extra spaces** anywhere in the returned string. The only possible spaces are ones that were already present in the strings passed in for the parameters.

2. Implement the method called `fieldsFor()` whose header we have provided. It takes a string representing the id number of a movie, and it should return a string containing a sequence of XML elements – one for each non-null field (i.e., column value) in that movie's tuple. If a field has a null value, it should **not** be included in the returned string. For example:

   ```
   > XMLforMovies xml = new XMLforMovies("movie.sqlite")
   > xml.fieldsFor(xml.idFor("Black Panther"))
   "    <name>Black Panther</name>
       <year>2018</year>
       <rating>PG-13</rating>
       <runtime>134</runtime>
   ```

```
    <genre>AVS</genre>
    <earnings_rank>3</earnings_rank>
"
> xml.fieldsFor(xml.idFor("West Side Story"))
"    <name>West Side Story</name>
    <year>1961</year>
    <runtime>151</runtime>
    <genre>DLR</genre>
"
> xml.fieldsFor("1234567")    // no movie with that id
""
```

In our database, *West Side Story* has null values for rating and earnings_rank, and thus those elements are not included in the string returned for that movie.

Important notes:

- You should begin by performing the appropriate SQL query. Use the `idFor()` method as a model for what you should do. If the query produces no results, the method should return an empty string, as shown in the third example above.
- The elements for the non-null fields should appear in the same order as the corresponding columns in the `Movie` table:
  - name
  - year
  - rating
  - runtime
  - genre
  - earnings_rank
- You **must** use your `simpleElem()` method to form the XML element for a given field.
- Each element should be on its own line, which you can accomplish by including a newline character (`"\n"`) after each element.
- Each element should be preceded by *exactly four spaces*, and there should be **no extra spaces** at the end of a given line. (Note: In the first two examples shown above, it looks like the first element is indented more, but that is an artifact of the double-quote character shown at the beginning of the string.)

3. Implement the method called `actorsFor()` whose header we have provided. It takes a string representing the id number of a movie, and it should return a string containing the XML for a single complex element named `actors` that includes nested child elements of type `actor` for all of the actors associated with the movie, ordered by the full name of the actor. For example:

```
> XMLforMovies xml = new XMLforMovies("movie.sqlite")
> xml.actorsFor(xml.idFor("Black Panther"))
"    <actors>
      <actor>Chadwick Boseman</actor>
      <actor>Danai Gurira</actor>
      <actor>Lupita Nyong'o</actor>
      <actor>Martin Freeman</actor>
      <actor>Michael B. Jordan</actor>
    </actors>
"
> xml.actorsFor(xml.idFor("Wonder Woman"))
"    <actors>
      <actor>Chris Pine</actor>
      <actor>Connie Nielsen</actor>
      <actor>Danny Huston</actor>
      <actor>Gal Gadot</actor>
      <actor>Robin Wright</actor>
    </actors>
"
> xml.actorsFor("1234567")    // no movie with that id
""
```

Important notes:

- You shouldn't make any assumptions about the number of actors associated with a given movie. In many cases it will be five, but your code should be able to handle an arbitrary number.
- Here again, you should begin by performing the appropriate SQL query and processing it using the appropriate JBDC method calls. Because you need to process an arbitrary number of tuples, the `createFile()` method provides a good model for what you should do. If the query produces no results, the method should return an empty string.
- Make sure to use an `ORDER BY` clause in your query so that the actors will be ordered by their full name, as shown above.
- You **must** use your `simpleElem()` method to form the XML child element for each actor.
- The returned string should be formatted as shown above. The outer start tag, the outer end tag, and each child element should be on its own line, with a newline character (`"\n"`) at the end of each line.
- The outer start tag and outer end tag should each be preceded by *exactly four spaces*, but the child elements should each be preceded by *exactly six spaces*. There should be **no extra spaces** at the end of a given line.

4. Implement the method called `directorsFor()` whose header we have provided. It takes a string representing the id number of a movie, and it should return a string containing the

XML for a single complex element named `directors` that includes nested child elements of type `director` for all of the directors associated with the movie, ordered by the full name of the director. For example:

```
> XMLforMovies xml = new XMLforMovies("movie.sqlite")
> xml.directorsFor(xml.idFor("Black Panther"))
"    <directors>
      <director>Ryan Coogler</director>
    </directors>
"
> xml.directorsFor(xml.idFor("Frozen"))
"    <directors>
      <director>Chris Buck</director>
      <director>Jennifer Lee</director>
    </directors>
"
> xml.directorsFor("1234567")    // no movie with that id
""
```

The guidelines for the previous method also apply here, but for `directors` and `director` elements instead of `actors` and `actor` elements.

5. Implement the method called `elementFor()` whose header we have provided. It takes a string representing the id number of a movie, and it should return a string containing the XML for a single complex element named `movie` that includes nested child elements for:

   - the non-null fields in the movie's tuple (use `fieldsFor()` to get these)
   - the actors in the movie (use `actorsFor()` to get these)
   - the directors of the movie (use `directorsFor()` to get these)

   For example:

```
> XMLforMovies xml = new XMLforMovies("movie.sqlite")
> xml.elementFor(xml.idFor("Black Panther"))
"  <movie id="1825683">
    <name>Black Panther</name>
    <year>2018</year>
    <rating>PG-13</rating>
    <runtime>134</runtime>
    <genre>AVS</genre>
    <earnings_rank>3</earnings_rank>
    <actors>
      <actor>Chadwick Boseman</actor>
      <actor>Danai Gurira</actor>
      <actor>Lupita Nyong'o</actor>
      <actor>Martin Freeman</actor>
```

```
        <actor>Michael B. Jordan</actor>
      </actors>
      <directors>
        <director>Ryan Coogler</director>
      </directors>
    </movie>
"
> xml.elementFor("1234567")   // no such movie
""
```

Important notes:

- This method should not perform any queries of its own. Rather, you must use your previous methods to obtain the necessary child elements.
- The start tag for the movie must include the movie's id as an attribute, as shown above. In order to include the quotes around the id, you will need to use the escape sequence `"\""` for each double-quote character.
- The outer start tag and outer end tag should each be on their own line preceded by *exactly two spaces* and followed by a newline character. The child elements should have the same spacing and formatting described in the earlier method specifications, so you won't need to add any new spaces to them. Once again, there should be **no extra spaces** at the end of a given line.

Once you have completed and tested all of your methods, running the program will create a file named `movies.xml` that represents the entire `Movie` table in XML!

Problem 8: Querying an XML database

20 points total; 5 points each part

This problems asks you to construct XPath and XQuery queries for an XML version of our entire movie database. **The schema of this XML database is described here**.

To allow you to check your work, we have loaded this database into a container that is managed by Berkeley DB XML, a native XML DBMS that we have included on the [virtual machine][vm] that you installed earlier in the semester.

Getting Started

To prepare for your work with Berkeley DB XML, you will need to take the following steps:

1. Start up the [virtual machine][vm].
2. Start up the Firefox browser on the [virtual machine][vm], and load this assignment in that browser.
3. Download the following files onto the [virtual machine][vm]:

- `imdb.dbxml`
- `ps3_queries.txt` (you can use a right-click followed by *Save Link As...* to download it rather than viewing it in the browser)

4. By default, the files will be downloaded into the `Downloads` directory within your home directory on the [virtual machine][vm]. Feel free to move them into a different directory if you prefer, as long as both files are stored in the same directory.

Performing queries in Berkeley DB XML

Once you have completed the steps outlined above, you can perform queries by taking the following steps:

1. `cd` into the directory in which you stored your copy of the `imdb.dbxml` file (see step 3 above).
2. Start up Berkeley DB XML from within that directory:

```
dbxml
```

   This will put you in the DBXML shell, which has the following command prompt:

```
dbxml>
```

3. Once you are in the DBXML shell, open the container in which the database is stored:

```
dbxml> openContainer imdb.dbxml
```

   (Note: You should **not** type in `dbxml>`. It should appear as a prompt when you are in the DBXML shell.)

4. To execute a query, begin with the keyword `query`, followed by an XPath expression or XQuery FLWOR expression surrounded by single quotes. For example:

```
dbxml> query 'for $m in collection("imdb.dbxml")//movie where $m/year = 1
```

   This should produce a message indicating the number of items in the results of the query:

```
8 objects returned for eager expression...
```

5. To see the results, enter a `print` command:

```
dbxml> print
```

For the query above, you should see the following results:

```
<name>Home Alone</name>
<name>Ghost</name>
<name>Dances with Wolves</name>
<name>Pretty Woman</name>
<name>Teenage Mutant Ninja Turtles</name>
<name>Reversal of Fortune</name>
<name>Misery</name>
<name>Goodfellas</name>
```

Important guidelines

1. If you are using a Mac, you should disable smart quotes, because they will lead to errors in Berkeley DB XML and in our testing. There are instructions for doing so here.
2. Construct the queries needed to solve the problems given below, and put your answers into the `ps3_queries.txt` file that you downloaded above. Use a text editor to edit the file, and make sure that you keep the file as a plain-text file.
3. Each of the problems must be solved by means of ***a single query*** (i.e., a single `query` command, followed by a single `print` command). Unless the problem specifies otherwise, you may use either a standalone XPath expression or an XQuery FLWOR expression. Use nested subqueries as needed.
4. Your queries should only use information provided in the problem itself. In addition, they should work for *any* XML database that follows the schema that we have specified.
5. For each problem, put the `query` and `print` commands in the space provided for that problem in the file (in between the second and third `echo` commands for that problem). We have included a sample query that you can use as a model. ***Do not remove or modify any of the other lines in the file.***
6. You should ***not*** include the results of the queries, but only the queries themselves.
7. Test each query in Berkeley DB XML to make sure that it works. You can do this through an interactive session as described above, or you can test the queries that are currently in your `ps3_queries.txt` files by doing the following from the Unix command line:

```
dbxml -s ps3_queries.txt
```

You can also send the results to a file as follows:

```
dbxml -s ps3_queries.txt > output.txt
```

8. Once you have added all of your queries to your `ps3_queries.txt` file, create an output file as described in the previous step, and make sure that the output file looks reasonable. Modify your queries as needed until you get the desired output.

The query problems

Make sure to read and follow the guidelines given above.

It is worth noting that our movie database was last updated in September of last year, so it doesn't include information about the most recent movies or Academy Award winners.

1. Write a ***standalone XPath expression*** (*not* a FLWOR expression) to find the names of all people in the database who will celebrate a birthday on the April 3 (the day this assignment is due). The result of your query should be a collection of `name` elements with the names of these people.

   Hints:

   - You should assume that all dates of birth are stored in the form `yyyy-mm-dd`. For example, April 3, 2019 would be stored as `2019-04-03`.
   - You should use the `contains` function, which performs substring matching. For example, we could use the following expression to obtain the `movie` elements for of all of the Harry Potter movies in the database:

     ```
     collection("imdb.dbxml")//movie[contains(name, "Harry Potter")]
     ```

2. Now write an XQuery FLWOR expression to find the same people as the previous query, ***but*** this time you should:

   - generate new elements of type `april-third` whose values consist of the name of the person, followed by a space, followed by the person's date of birth surrounded by parentheses. For example:

     ```
     <april-third>Marlon Brando (1924-04-03)</april-third>
     ```

   - order the results by date of birth, starting with the person whose date of birth is furthest in the past.

   Hints:

   - You will need to use the `text()` function to obtain the values of the relevant elements, without their start and end tags.
   - You will also need to use braces and commas as part of your `return` clause. See the examples in the lecture notes for models of how to transform the elements selected by a query into new types of elements.

- The `return` clause can also include string literals (e.g., `" ("`).

3. Find all people who have won an Oscar two years in a row. (The types of the two Oscars don't matter; all we care about is that they were won by the same person in consecutive years.) The result of your query should be one or more new elements of type `back-to-back` that consist of the following:

   - the `name` child element of the person
   - new child elements called `first-win` and `second-win` whose values consist of the type and year of the first Oscar and second Oscar respectively. In both cases, there should be a single space between the type and year and parentheses surrounding the year.

   For example:

   ```
   <back-to-back>
     <name>Alejandro Gonzalez Inarritu</name>
     <first-win>BEST-DIRECTOR (2015)</first-win>
     <second-win>BEST-DIRECTOR (2016)</second-win>
   </back-to-back>
   ```

   Hints:

   - The hints for the previous query problem also apply to this problem.
   - Predicates can include arithmetic operators, and this should allow you to ensure that the Oscars are from consecutive years.
   - To make your output easier to read, you should include string literals with newline characters (`\n`) that force the child elements to be on separate lines and indented slightly. Here is an example of doing this in the context of another query:

     ```
     for $m in collection("imdb.dbxml")//movie
     where contains($m/name, "Harry Potter")
     return <potter-film>
            {
             "\n  ", $m/name,
             "\n  ", $m/year,
             "\n"
            }
            </potter-film>
     ```

4. For each of the movie ratings in the database, construct a new element of type `rating-info` that includes the following:

   - a child element called `rating` that contains the rating name; see below for more details about this element

- a new child element of type `num-movies` that has as its value the number of movies in the database with that rating
- a new child element of type `avg-runtime` that has as its value the average runtime of movies in the datbase with that rating.
- 0 or more child elements of type `top-ten`, each of which contains the name of a movie with that rating that is among the top **10** grossing movies of all time, according to its `earnings_rank`.

For example:

```
<rating-info>
  <rating>PG</rating>
  <num-movies>140</num-movies>
  <avg-runtime>112.33571428571428571428571143</avg-runtime>
  <top-ten>Incredibles 2</top-ten>
</rating-info>
```

Hints:

- To ensure that you only consider a given rating once, you should use the `distinct-values` function. For example, to iterate over distinct Oscar types, you could do the following:

```
for $t in distinct-values(collection("imdb.dbxml")//oscar/type)
```

- `distinct-values` gives you the *text* values of the corresponding elements, not the elements themselves. As a result, your `return` clause will need to construct new `rating` elements by adding back in the begin and end tags that were removed by `distinct-values`.
- You can use the built-in `count()` and `avg()` functions to compute the necessary values.
- You will need to use a nested FLWOR expression.
- Several of the hints for the previous problems also apply here.

Submitting your work for Part II

Login to Gradescope by clicking the following link: Gradescope

Once you are in logged in, click on the box for ***CS 460***.

You will make submissions to ***two separate assignments*** on Gradescope. The steps needed for the two submissions are different, so please make sure to follow carefully the procedures outlined below.

PS 3, Problem 6

Submit your `ps3pr6.pdf` file using these steps:

1. If you still need to create a PDF file, open your file on Google Drive, choose *File->Download as->PDF document*, and save the PDF file on your machine.

2. Click on the name of the assignment in the list of assignments on Gradescope. You should see a pop-up window labeled ***Submit Assignment***. (If you don't see it, click the ***Submit*** or ***Resubmit*** button at the bottom of the page.)

3. Click the *Select PDF* button and find the PDF file that you created. Then click the *Upload PDF* button.

4. You should see a box saying that your submission was successful. Click the `(x)` button to close that box.

5. You should see a question outline along with the contents of your uploaded PDF. ***Make sure that the contents contain the work that you want us to grade.***

6. If needed, use the ***Resubmit*** button at the bottom of the page to resubmit your work.

PS 3, Problems 7 and 8

Submit your `XMLforMovies.java` and `ps3_queries.txt` files using these steps:

1. Click on the name of the assignment in the list of assignments. You should see a pop-up window with a box labeled ***DRAG & DROP***. (If you don't see it, click the ***Submit*** or ***Resubmit*** button at the bottom of the page.)

2. Add your files to the box labeled ***DRAG & DROP***. You can either drag and drop the files from their folder into the box, or you can click on the box itself and browse for the files.

3. Click the ***Upload*** button.

4. You should see a box saying that your submission was successful. Click the `(x)` button to close that box.

5. The autograder will perform some tests on your files. **Once it is done, check the results to ensure that the tests were passed.** If one or more of the tests did *not* pass, the name of that test will be in red, and there should be a message describing the failure. Based on those messages, make any necessary changes. Feel free to ask a staff member for help.

   Note: You will **not** see a complete Autograder score when you submit. That is because additional tests for at least some of the problems will be run later, after the final deadline for the submission has passed. For such problems, it is important to realize that passing all of the initial tests does **not** necessarily mean that you will ultimately get full credit on the problem. You should always run your own tests to convince yourself that the logic of your solutions is correct.

6. If needed, use the ***Resubmit*** button at the bottom of the page to resubmit your work. ***Important:*** **Every time that you make a submission, you should submit *all* of the files for**

that Gradescope assignment, even if some of them have not changed since your last submission.

7. Near the top of the page, click on the box labeled ***Code***. Then click on the name of each file to view its contents. Check to make sure that the files contain the code that you want us to grade.

Important

- It is your responsibility to ensure that the correct version of every file is on Gradescope before the final deadline. ***We will not accept any file after the submission window for a given assignment has closed, so please check your submissions carefully using the steps outlined above.***
- If you are unable to access Gradescope and there is enough time to do so, wait an hour or two and then try again. If you are unable to submit and it is close to the deadline, email your homework ***before the deadline*** to `cs460-staff@cs.bu.edu`