Homework 5: Query Execution Engine v0.1

Due: 5:00 PM, November 24, 2014

For this assignment, you will be working again with records, pages, and files. This time, however, we will allow records to contain data from any defined schema type and we will introduce a relational operator.

Ex 1 (30 points): Formalizing Data

The data that you will be managing will be of type `Record` (a class you will create) and will support data from any Schema which uses types: String, Integer, and Double. That is, a `Record` will correspond to a row of data from one of potentially an infinite number of schemas.

Your code *must* define the following `Record` class, and it can be implemented in any way you wish. The only stipulation is that you must override the [`toString()`] method or overload [`friend ostream& operator<< (ostream& os, const Record& r)`] so that I can view the contents of each `Record` that your application produces. Therefore – in its minimal form – your `Record` class must look like this.

```
public class Record {
  // Write code to store/handle an arbitrary number of
  //    - Attribute Names
  //    - Attribute Types
  //    - Attribute Values

  public String toString() {
    // Implement this to return a string displaying all attribute values
  }
}
```

or

```
class Record {
  // Write code to store/handle an arbitrary number of
  //    - Attribute Names
  //    - Attribute Types
  //    - Attribute Values
public:
  friend ostream& operator<< (ostream& os, const Record& r) {
    // Implement this to write a string to os displaying all attribute values
  }
}
```

As stated in the code above, for each attribute a `Record` contains, a `Record` must store: 1) an attribute name; 2) an attribute type (being `String`, `Double`, or `Integer`); and 3) an attribute value.

Ex 2 (30 points): Reading in Data

Once you have created your `Record` class you should begin to write a class that will read a file from disk one page at a time (where a page is of type: `Vector<Record>`) in order to pass the data on to operators to be processed. A sample of the file format can be seen below:

```
FirstName,LastName,Year,GPA
String,String,String,Double
---------------------------
Chris,Knight,Fr,3.8
Mitch,Taylor,Jr,3.6
Lazlo,Hollyfeld,Sr,4.0
Paul,Stephens,Sr,3.2
David,Lightman,Jr,2.8
Stephen,Falken,Jr,3.7
Joshua,Falken,Fr,3.2
Dade,Murphy,So,3.3
Kate,Libby,Jr,3.9
```

The first line of a file will contain all of the attribute names for the attributes in the file. There will be a comma between each attribute name and attribute names will not contain spaces. The following line in the file will contain the types of the corresponding attributes. The only types you should be prepared to receive are: `String`, `Double`, and `Integer`. Note, that these type names are case sensitive (exactly as they are in Java). The following line can be discarded and is simply used to visually distinguish the schema from the data. The remaining lines in the file will each represent a record containing attribute values in the same order as they were observed in the schema. Each record will always be on its own line and there will not be any blank lines in the file. Trust that the input will be well-formed without any surprises to trip you up. On that note, you can be assured that commas will only appear in the data as delimiters. Therefore, you will likely find the Java `split()` function very handy to parse your attributes from each line of text.

To read in the files from disk I, personally, made a `FileReader` class. This class looked very similar to the `TaxRecordReader` class that I wrote for the previous assignment. Like the `TaxRecordReader` class, it also has functions called `Open(String fileName, int pageSize)` and `Close()` which are quite similar – the difference being that my `FileReader` class also parsed the file schema in the `Open()` function. I re-purposed and renamed the `ReadPage()` function from the `TaxRecordReader` to be the `Next()` function in my `FileReader` class which returns data of type: `Vector<Record>`. I renamed the `ReadPage` method to `Next()` so that it would be in-line with the `Next()` method defined in the operator iterator model which is used to fetch pages of records from operators (more on this in exercise 2).

In summary, write a class that reads data of type `Record` from the file format described above to produce pages of data at a time (`Vector<Record>`).

Ex 3 (40 points): Defining and Using Operators

Now that you can store and read data, it's time to do something slightly interesting with it. In this exercise you will implement an operator that uses the standard iterator model of `Open()`, `Next()`, and `Close()`. To do this, you should start with this abstract class which defines a specific means for interacting with sub-classes (namely the iterator model):

```
public abstract class Operator {

  // Project Open interfaces
  public boolean Open(String fileName, Vector<String> fieldNames) { return false; }
  public boolean Open(Operator op,     Vector<String> fieldNames) { return false; }

  public abstract Vector<Record> Next();
  public abstract void Close();

  protected int pageSize = 10;
}
```

Note that we hard-code the `pageSize` here. Since this will be a superclass of all other operators, then each operator class you implement that `extends Operator` will be able to access `pageSize` as if it was a public member of the base class. Also note that `Project` will support two `Open` methods. One takes a file as its input source (presumably activating your `FileReader` class – hint, hint...) and the other takes another operator as its input source (and eventually calling the `Next()` method on it to fetch data from it).

You should then implement the `Project` class to provide the functionality of the project operator. It should start out looking something like this:

```
public class Project extends Operator {

  public boolean Open(String fileName, Vector<String> fieldNames) {
    // Implement this
  }
  public boolean Open(Operator op, Vector<String> fieldNames) {
    // Implement this
  }

  public Vector<Record> Next() {
    // Implement this

    // Return an empty Vector if no more pages exist
  }

  public void Close() {
    // Implement this
  }
}
```

Observe that we're seeing a little dash of *polymorphism* here and in the `Operator` super-class. They feature an `Open()` method which takes an `Operator` as its input source. Therefore, it may actually be passed any sub-class that `extends Operator`. Since each of these sub-classes will each implement the `Next()` method then it is possible to simply call `op.Next()` regardless of what type of operator is being used as the source. In my code, I used an `interface` called `INext` which defined the existence of the `Next()` method (and was implemented by both `Operator` and `FileReader`) to seamlessly acquire data via the `Next()` method for my source regardless of whether that source was an `Operator` sub-class or my `FileReader`. This is not required, but it sure did cut down on a lot of extra/redundant code on my part!

Once you have finished with this portion you should be able to test out your code! I sure will. This can be done using code similar to the following and using the (student.data) text file used as a previous example:

```
Vector<String> names = new Vector<String>();
names.add("FirstName");
names.add("LastName");

Project op1 = new Project();
op1.Open("student.data", names);

Vector<Record> page;
do {
  page = op1.Next();
  for (Record r : page) {
      System.out.println(r.toString());
  }
} while (page.size() > 0);

op1.Close();
```

The idea is that you continue to call `Next()` on operator at the top of your query tree untill all data have been processed and produced all the way through the tree. Make sure that this code works on your computer as I will be using similar code to test your application. So, I do expect this interface to work exactly as it's shown here.

Submission: Turning in the assignment

To turn in your assignment, place your work on the CS-Data server [redacted] under the directory within your student folder: `cmsc321\hw_05`. Make sure your folder structure is identified **exactly** as it is outlined here or you may lose points for not following directions and hindering my scripts which run your applications.