CS 4320/5320 Homework 4

Spring 2016

Due April 13th, 2016 at 11:59 pm

This assignment is out of 68 points and counts for 10% of your overall grade.

This assignment, especially the Hadoop portion, requires **significant installation and configuration**. For this reason, it is essential that you start early. In particular, if you are not familiar with Linux and/or command line work, you should get started over Spring Break and at least attempt installation following the instructions provided.

1 PageRank using MapReduce on Hadoop (38 points)

In this part you will be using Hadoop to implement PageRank using MapReduce. Please read all the material below before you jump into coding. It is a lot of information and you may want to go through it more than once before you get started.

In the below instructions, we first explain how to install and run the correct Hadoop environment on Linux, then we explain what you need to implement, and finally we give a tour of the auxiliary classes that we have provided.

1.1 Installation

Platform. This assignment was written and tested in Ubuntu, and we recommend that you complete it on the same. **We are only officially supporting Linux, specifically Ubuntu, for this assignment.** While you are welcome to attempt this assignment on Windows or Mac, and we will assist you to the best of our ability, we cannot guarantee we will be able to help you on these other platforms.

If you do not have Linux but wish to use it, you can use virtualization. A couple of free options are:

1. VirtualBox, available free online at `https://www.virtualbox.org/wiki/Downloads`.
2. VMware. If you are a Cornell CIS student, you can get this free to students at OnTheHub (`https://e5.onthehub.com/WebStore/Welcome.aspx?ws=9cf5631b-842e-de11-a497-0030485a8df0`). If you use VMware, you will have to create a new virtual machine and install Ubuntu on it. The installation is fairly simple and you can use your favorite search engine for help with installation.

The remainder of the instructions will proceed on Ubuntu.

Build tools. Some familiarity with command line interfaces will be required. If you are not familiar with command line interfaces, you may use your favorite search engine to discover how to use a terminal. This link may be of some help:
`http://www.pas.rochester.edu/~pavone/particle-www/telescopes/ComputerCommands.htm`. In Linux, you can access the manual for a command by typing `man <command name>` into the terminal.

Open a terminal and run:

```
sudo apt-get install build-essential
```

As the name implies, this will install build software, particularly `make`. You can verify installation by running

```
make -v
```

You should get something that looks like a version and not a "command not found" error.

Java. We will be using Java 7 for this assignment. **Do not use Java 8 functionality.**

To install Java 7, run:

```
sudo apt-get install openjdk-7-jre openjdk-7-jdk
```

You can verify installation by running

```
javac -version
java -version
```

Again, you should get something that looks like a version and not a "command not found" error.

Hadoop. Hadoop can be downloaded via web browser from its release page at
`https://hadoop.apache.org/releases.html`. We will be using the latest version at time of writing, which is 2.7.2. You want the "binary" version under the "Tarball" column.

Once the download is complete, open a terminal and change your directory to the location where it was downloaded. This is most likely your Downloads directory:

```
cd ~/Downloads
```

Extract the contents by typing the following command into the terminal:

```
tar -xvf hadoop-2.7.2.tar.gz
```

To keep these instructions simple and consistent, we will install Hadoop to the `/opt/` folder. You (or an installer on your behalf) can, however, put it wherever you want. If you do so, replace all instances of `/opt/` in these instructions to your path. You will also need to edit `HADOOP_DIR` in the `Makefile` to point to where the actual installation is.

Move the uncompressed directory to `/opt/` by running the command:

```
sudo mv hadoop-2.7.2 /opt/
```

Verify that you see a directory at `/opt/hadoop-2.7.2/bin/`. You will need to add the whole path to this directory to your `PATH` environment variable, so you can run Hadoop from any directory. (The whole path can be obtained by stepping into the directory and typing pwd). To modify `PATH`, open `~/.bashrc` with a graphical text editor by typing: `gedit ~/.bashrc`. Add the following line to the very bottom of the file:

```
export PATH=$PATH:/opt/hadoop-2.7.2/bin
```

Also, while the file is open, if `JAVA_HOME` does not already exist, add this line to the very bottom:

```
export JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk-amd64
```

The value on the right side of "=" is the path to your Java folder. Note that the suffix ("amd64") may differ depending on your processor's architecture; you may want to check the contents of the directory before adding the line using

```
ls /usr/lib/jvm/
```

Save and exit the file. Quit and reopen the terminal (or type `source ~/.bashrc`) and type `echo $JAVA_HOME`. Make sure the Java path you entered appears.

Verify that Hadoop was installed correctly using the command : `hadoop version`. As before, you should get a descriptive output and not a "command not found" error.

As a way to get started with Hadoop, you may want to look at this tutorial: `http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html`

1.2 Getting started with the code

Download the zip file that we have provided and place it in a folder of your choice. In that folder run the command: `unzip page_rank.zip` You will see a folder called `page_rank`. Run the command: `cd page_rank` to step inside the folder. You will see the following contents:

- `Makefile`
- `src` folder
- `tests` folder

- `expected` folder: This holds the results to the sample tests provided in the `tests` folder.

Run the `Makefile` using the command `make &>log.txt` When it's done, if you look at the directory, you will see directories named `stage0` to `stage9`, a file called `log.txt`, and a jar file `PageRank.jar`. You have just compiled and run the skeleton code! This is a Hadoop MapReduce job that does nothing.

If this does not happen, check the log file for any errors. Also make sure that the `HADOOP_DIR` in the `Makefile` actually points to your Hadoop installation.

If you look at the `log.txt` file (open it using a text editor), you'll see a lot of information about the Hadoop run. This would have normally been printed to the screen, but the `&>log.txt` puts it in a file named `log.txt`. For debugging, we recommend that you put `System.out.println()` calls into your code. However, instead of being displayed on the screen, they will be put in `log.txt` if you run the above command.

Before you can run the code again, you'll need to run the command `make clean`. This will clean up everything from the previous run. However, it won't delete your log file. Now, you can start implementing PageRank!

The textbook at `http://lintool.github.io/MapReduceAlgorithms/ed1n/MapReduce-algorithms.pdf` contains a description of Page Rank as well as pseudocode in section 5.3. We recommend that you go through this.

Implementing PageRank In the skeleton code, there are five files that you should be changing. You will be implementing two mappers and two reducers, which will work together to implement the PageRank algorithm. You will also be adding a little code to the main PageRank class. The five files are:

- `TrustMapper.java`
- `TrustReducer.java`
- `LeftoverMapper.java`
- `LeftoverReducer.java`
- `PageRank.java`

The `TrustMapper` and `TrustReducer` represent the core of the PageRank algorithm: they propagate the PageRank of nodes into their out-neighbors. The `LeftoverMapper` and `LeftoverReducer` tie up the loose ends. The PageRank that was lost from nodes with no out-links is spread evenly across all nodes, and the random jump factor is taken care of. (See Formula 5.2 in the textbook.)

To implement these, you will need to correctly use the framework of Hadoop. Much of that framework has been built up for you, as you can see below. However, you'll need to use the Hadoop API in your program. You can find that reference at `http://hadoop.apache.org/docs/stable/api/`.

The PageRank class is mostly written for you. However, you'll find one part that has a comment:

```
//set up leftover and size
```

This relates to keeping track of "lost" PageRank from nodes with no outlinks. At this point in the file, you should add code to set up the variables correctly so that they can be passed into the next mapper/reducer

pair. We recommend using a Hadoop counter to accomplish this, but you may choose a different implementation.

If you do choose to use a counter for this, you will need to create a new .java file that contains an enum. Then, you will need to include that in the `Makefile`. If you look in the `Makefile`, you will see that there is a line:

```
bases = LeftoverMapper LeftoverReducer NodeInputFormat Node NodeOrDouble NodeOutputFormat
NodeRecordReader NodeRecordWriter TrustMapper TrustReducer PageRank
```

You will need to change this to add the name of your new file (without the `.java`) to the end of the list. This will automatically add it to the build process that you need to run your project in Hadoop.

1.3 Code tour

The skeleton code that we have provided you with does most of the Hadoop work for you. However, to implement PageRank, you will have to interface with this code. This means that it's going to be important that you know what this code does. **IT IS HIGHLY RECOMMENDED THAT YOU DO NOT EDIT THIS CODE!**

Much of the code is somewhat fragile, and it will be difficult to get working again if you change it. It is very possible to solve the problem with the skeleton code you have been given. Now, we will begin to look at the different Java files you have been given. Feel free to look at the code inside the files and read along.

PageRank.java. `PageRank` is the main class of your program. It sets up the Hadoop job and tells Hadoop where to read from and write to. It consists of 4 methods: the main method and 3 methods that return `Job` objects. The `Job` class represents a MapReduce run in Hadoop. The three methods that return `Job` set up a properly-formatted run. The main method is much more interesting. Here, we actually run the correct job. The idea is that, every other run, we will run the `TrustMapper/TrustReducer` job, and on the odd runs we run the `LeftoverMapper/LeftoverReducer` job. For the first job, the input comes from the input directory. From then on, the input comes from the output of the last run.

Node.java. `Node` is probably the class you will deal with the most. It represents a node in the internet graph (i.e. a web page). It contains an identification number, a PageRank value, and a collection of outgoing links. It implements the iterable interface, which allows you to use a for-each loop to go through the outgoing links. The write and readFields methods are from the `Writable` interface. `Writable` allows for objects of class Node to be used as output and input from `Mapper`s and `Reducer`s.

NodeOrDouble.java. `NodeOrDouble` is a class that may seem a bit strange to you on first glance. You may construct it with either a `Node` object or a `Double` object. Afterwards, you cannot change which your `NodeOrDouble` object holds. You can, however, tell which it holds (`isNode`) and get out the object it holds (`getNode` and `getDouble`). Be careful, trying to retrieve the wrong type will result in a `null` object! Again, it implements the `Writable` interface.

NodeInputFormat.java and NodeRecordReader.java. These classes provide the ability to read from files and get a `Node` out. This means that you don't have to parse lines from the text file yourself. The `NodeInputFormat` class tells Hadoop how to create an appropriate `RecordReader`, in our case the

`NodeRecordReader`. It doesn't do anything else besides return a constructor for a `NodeRecordReader`. The `NodeRecordReader` class parses a file that contains `Node`s for you. A line in a node file looks like:

```
nodeid pagerank out1,out2,...
```

where `nodeid` is an integer, `pagerank` is a decimal number, and `out1, out2, ...` are all integers. Here, `nodeid` is the id of the node, `pagerank` is the node's current PageRank value, and the `out`s are the outgoing links of the node.

NodeOutputFormat.java and NodeRecordWriter.java. These classes are the inverse of `NodeInputFormat` and `NodeRecordReader`. They print out `Node`s to a file in the same format as above. They allow Hadoop to simply output nodes, without you having to do any pretty printing in your mapper and reducer code.

Makefile. This is not a Java file, as you can probably guess. Makefiles are designed to automate the compiling, running, and cleanup of large projects. Since Hadoop requires quite a bit to get set up, we've automated this with a Makefile. As described earlier, the command `make` will compile your Java and run Hadoop for you. The command `make clean` will then clean up everything from that process for you. For your purposes:

1. If you run `make`, your code should compile and run and generate several intermediate directories.
2. If you run `make clean`, your intermediate directories will be cleaned up.

If you're interested in `make`, please see the official documentation at `http://www.gnu.org/software/make/manual/html_node/index.html`.

Tests. To test a file, place one (1) of the tests in the input directory and type `make clean`. If you run `make clean` when no cleanup is needed (no intermediate directories are present), you will get errors which you may ignore. In order to get your code running, type: `make &>log.txt`.

Upon running this, your final result will be captured in `stage9/output.txt`. View this file before you run make clean again, so that you know what your final page rank values are. We have given you a limited number of tests to play with. However, you should expect that your code will have to pass more tests. The goal is to match our reference implementation. Note that, because we are dealing with floating-point numbers, your answers may be close but not match ours exactly. In this case, you can be considered to be passing the test. You can find our example output in the "expected" folder that is provided to you inside the zipped archive.

There are four sample tests we have provided. The test `center` has one central node that all others are connected to, and other structure among the nodes. The test `singleton` is just a single node with no connections. `triangle_cycle` is three nodes arranged in a cycle. In the `line` test, the first node connects to all the others, the second to all the others except the first one, and so on (the last node has no out-edges).

1.4 Grading

We will be running your submission on the above four (4) sample test cases as well as seven (7) undisclosed test cases for a total of eleven (11) test cases. Each test case is worth three (3) points. Five (5) additional points are allocated for coding style.

2 Graph queries using Neo4j (30 points)

2.1 Installation

You can get Neo4j from `http://neo4j.com/download/`. The Community Edition will be sufficient. We will be using version 2.3.2. Note that syntax etc. has been known to change between versions, so we recommend that you use this version.

Some useful documentation:

- Manual: `http://neo4j.com/docs/stable/index.html`.
- Reference card: `http://neo4j.com/docs/stable/cypher-refcard/`

2.2 Data model

The data model for this part simulates a role-playing game. There are a bunch of characters such as a `Dragon`, each of which has a set of abilities that they can perform, such as `Fly` or `Fire Breath`. Each character and each ability has an element associated with it, such as `Air` or `Fire`. Finally, some elements may be especially effective against each other, e.g. `Fire` is effective against `Air`. See Figure 1 for an example.

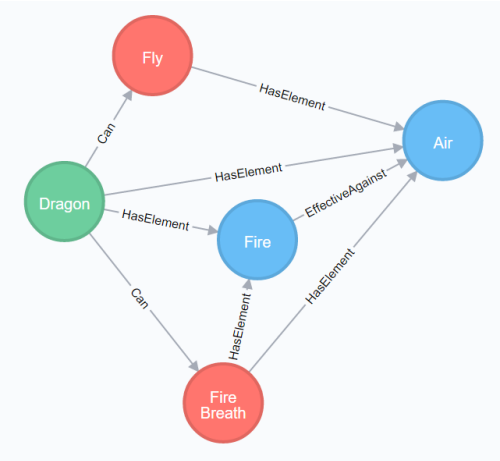

Figure 1: Example subgraph. The actual database will of course be larger than this.

More formally:

- There are three types of nodes: `Character`s, `Ability`s, and `Element`s.
- Each `Character` may have `Can` relationships to any number of `Ability`s.
- Both `Character`s and `Ability`s may have `HasElement` relationships to any number of `Element`s.
- Each `Element` may have `EffectiveAgainst` relationships to any number of other `Element`s.
- All nodes have a `name` property, which may be assumed to be unique. Specifically:

```
CREATE CONSTRAINT ON (c:Character)  ASSERT c.name IS UNIQUE;
CREATE CONSTRAINT ON (a:Ability)    ASSERT a.name IS UNIQUE;
CREATE CONSTRAINT ON (e:Element)    ASSERT e.name IS UNIQUE;
```

2.3 Exercises

Constraints. Your queries must follow these constraints:

- Your queries may only contain a single RETURN statement.
- Your queries must not modify the database. This includes, but is not limited to, using the keywords `CREATE`, `SET`, `UPDATE`, and `DELETE`. If these keywords appear in your file, you will get a 0 for this entire portion.

Again, **any queries that do not satisfy these constraints will receive a score of 0**.

Write queries to find the following:

Query 1. All pairs (character, `Fire`-element ability that the character can do). In our example database above this query would produce a single row (`Dragon`, `Fire Breath`). Order the result by the character name, then by the ability name.

Query 2. The names of all characters that cannot perform any non-`Physical`-element abilities. In other words, you want all characters with the following property: out of all the abilities the character can perform, each of those abilities is only associated with the `Physical` element or with no elements at all. Order the result by the character name.

Query 3. The name of the element such that the greatest number of characters can do at least one ability associated with that element. Return the element name and the number of characters. If there is a tie, return the tied element that is first alphabetically.

Query 4. For every pair of characters, the number of (distinct) elements that the characters' abilities have. That is, consider all the abilities of both characters, and consider all the elements associated with that (joint) set of abilities. We want the size of that set of elements.

Return the name of the first character, the name of the second character, and the number of (distinct) elements. Order the result by this number in descending order, then by the name of the first character, then by the name of the second character. Return only the top ten (10) results.

Do not pair characters with themselves. List each pair of characters once regardless of order. For example, include either (`Athena`, `Zeus`, `n`) or (`Zeus`, `Athena`, `n`), but not both.

Query 5. For each element, the collection of abilities that are associated at least one element which is `EffectiveAgainst` that element. In our example database, the `Air` row would contain a collection of all `Fire` abilities, since `Fire` is `EffectiveAgainst Air`. Note that in general several elements may be effective against a given element, so you would need to collate the abilities across all effective elements.

Return the name of each element and the list of abilities. The rows should be ordered by the name of the element, and each collection should be ordered by the name of the ability.

2.4 Grading

No partial credit will be given for code with syntax errors or code that produces incorrect results. Don't forget to add semicolons to the end of every query: we may paste the entire file at once for testing, and this would be a silly reason to get 0 points. Similarly, if you want to make notes to yourself, please do so using comments (`//`).

Each of the five (5) queries is worth five (5) points. Five (5) additional points are allocated for coding style.

3 Submission

3.1 Hadoop

Submit a .zip archive containing the following:

- Your `src` directory (and all `.java` files therein, whether you have modified them or not).
- Your `Makefile`.
- A `README` containing any special comments for the graders.

3.2 Neo4j

Submit a text file `queries.txt` containing your five queries. For your convenience, we have provided a skeleton file.

As a reminder, make sure that the queries are in the right order, and that any notes are done using comments (`//`).

3.3 Acknowledgement

If you consulted any outside sources, submit an `acknowledgement.txt` file.