Problem Set 4

All problems are due by 11:59 p.m. on Wednesday, April 17, 2019.

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

40 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: ps4_partI
2. Select *File->Make a copy...*, and save the copy to your Google Drive using the name `ps4_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 (`ps4_partI.pdf`) is the one that you will submit. See the submission guidelines at the end of Part I.

Problem 1: Replication

30 points total

Assume that a database is replicated across 7 different sites.

1. If the system uses ***fully distributed locking***, which of the following voting schemes (if any) would work? Explain briefly why each scheme would or would not work.
   a. update 6 copies, read 3 copies
   b. update 2 copies, read 6 copies
   c. update 5 copies, read 2 copies
   d. update 4 copies, read 4 copies
2. Now assume that the system uses ***primary-copy locking***. Which of the following voting schemes (if any) would work? Explain briefly why each scheme would or would not work.
   a. update 6 copies, read 3 copies
   b. update 2 copies, read 6 copies
   c. update 5 copies, read 2 copies
   d. update 4 copies, read 4 copies
3. If your workload primarily involves reads, which of the above configurations (1a, 1b, 1c, 1d, 2a, 2b, 2c, or 2d) would be the best one to choose? Explain briefly why you think it would be best, and make sure to also mention any potential drawbacks that it could have.

4. If your workload primarily involves writes, which of the above configurations (1a, 1b, 1c, 1d, 2a, 2b, 2c, or 2d) would be the best one to choose? Explain briefly why you think it would be best, and make sure to also mention any potential drawbacks that it could have.

Problem 2: Distributed locking with update locks

10 points

In lecture, we showed how to create global shared and exclusive locks from local locks of those types. What if we wanted to include not just shared and exclusive locks, but update locks as well, as discussed in the notes on concurrency control? Explain how you could create global shared, exclusive, and update locks from local locks of the same types. These global locks should have the same semantics as the equivalent locks in a non-distributed database in terms of when a lock can be acquired and how many transactions can hold a given type of lock at the same time.

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 (`ps4_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 4, 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 `ps4_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

60 points total

Overview

In this assignment, you will write Java programs to run MapReduce jobs using Apache Hadoop. For the problem domain, you will use a social network database that contains information about users and their connections to other users in the network.

The data is stored in a plain text file, where each line of the text file represents a single user. Here's an example line from the file:

```
18,Brown,Matthew,1989-11-05,mbrown@gmail.com,189,305,17,31;569121,235708,320
```

From left to right, each line contains the following fields:

- the user's unique ID
- the user's last name
- the user's first name
- the user's birthdate
- *optionally*, the user's email address
- *optionally*, a list of group IDs, separated by commas, specifying the groups the user belongs to
- *optionally*, a list of user IDs, separated by commas, referring to the user's friends

The fields are comma-separated, with the exception of a semicolon that appears before the friend list. However, if a user has no friends, the line will not contain a semicolon.

Here's a subset of lines from the text file, showing a number of different (though not all) variations:

```
65,Lewis,James,1965-12-26,jlewis1965@hotmail.com,257,7,227;911255,176121,554
80,Lawson,Emerald,1997-10-01,270,144,368,201,484,8;40146,44545,285685,734547
201,White,Alexander,1958-08-19,awhite1958@yahoo.com;979442,33777,416988,8234
```

Notes:

- You will need to determine how to correctly parse each line in the input file. In particular, you will need to account for the optional fields in any line. We encourage you to use the following methods as needed:
  - the `String` method called `split()` that we discussed in lecture

    - other `String` methods like `indexOf()` and `substring()`
    - the following static methods for converting from a Java `String` object that is the string representation of a number to the corresponding numeric value:
        - `Integer.parseInt()`
        - `Long.parseLong()`
        - `Double.parseDouble()`
- You may assume that the birthdate and email fields are well-formed; that is, the birthdate will always be of the form *YYYY-MM-DD*, where *YYYY* is the four-digit birth year, *MM* is the two-digit month, and *DD* is the two-digit day. The email, if present, will contain an occurence of the `@` character to separate the user name from the domain.

Resources

- the full Java API
- documentation for DrJava
- the Hadoop MapReduce API

Using MapReduce on Apache Hadoop

Apache Hadoop is already installed on the virtual machine that you used in PS 2 and PS 3. If for some reason you need to redownload the VM, you should do so as soon as possible.

We will be using Hadoop in *local* mode, where we simulate a distributed cluster locally on the VM. However, the queries that you construct could also be run on a real cluster without modification.

Getting started

Once you have started the VM, you should take the following steps to obtain the files that you will need for this assignment:

1. Run Firefox on the VM, and navigate to the page for this assignment (this page!).
2. Right-click on the following link, choose *Save Link As*, and save the file on your Desktop:

   `ps4.zip`
3. Find the downloaded file on your Desktop and double-click on it.
4. In the resulting window, click on the *Extract* button. This will open another window, and you should then click on its *Extract* button.
5. Once the files have been extracted, close any windows related to the extraction process and return to the Desktop, where you should see a folder named `ps4`. It contains:

- `WordCount.java` - a sample MapReduce program that you are welcome to use as a template for your programs
- starter code for each of the problems (`Problem3.java`, `Problem4.java`, etc.)
- `Makefile` - a file that will facilitate the compiling and running of your programs
- `input` - a folder containing the input data file for the problems you will solve
- `extra_data` - a folder containing two smaller versions of the input file, in case the original one ends up being too big in some cases
- a couple of other files that you will use to configure the system

All of your files for Part II should remain in the `ps4` folder.

Initializing Hadoop

The VM needs some additional configuration to be able to run the programs in local mode. Open the Terminal program on the VM and change directory to the `ps4` folder:

```
cd Desktop/ps4
```

Then copy two configuration files to the installation of Hadoop:

```
cp core-site.xml ~/.e66libs/hadoop-2.7.3/etc/hadoop
cp hdfs-site.xml ~/.e66libs/hadoop-2.7.3/etc/hadoop
```

Sample MapReduce program

In the `ps4` folder, we've given you a sample program called `WordCount.java`. It processes one or more text files, and it outputs a count for each word that appears in the files.

We've also given you a `Makefile` that will make it easier to compile and run your programs.

To compile and run `WordCount.java`, enter the following command from within the `ps4` folder:

```
make wc
```

When you do so, you'll see a series of commands that the `Makefile` executes on your behalf, along with any messages that are produced by those commands. (Note: It can take awhile for all of the steps to occur, so be patient!)

After the process has completed, you should see that a folder called `output_wc` has been created. It contains the results produced by the `WordCount` program.

To see the results, use the `cat` command on the output file, which is named `part-r-00000`:

```
cat output_wc/part-r-00000
```

Compiling and running your programs

1. Each program that you write for Part II should go in the `.java` file that we've provided for that problem. Make sure that you keep all of these files in the `ps4` folder.
2. To compile your program for a given problem without running it, enter the following command:

   ```
   make pN
   ```

   where you replace `N` with the number of the problem. For example:

   ```
   make p3
   ```

   will attempt to compile `Problem3.java`.
3. To run your program for a given problem (compiling it first if needed), enter the following command:

   ```
   make outputN
   ```

   where you replace `N` with the number of the problem. For example:

   ```
   make output3
   ```

   will attempt to run `Problem3.java`.

   Note: You may see one or more warning messages from the Hadoop system when it begins to run. That is to be expected.

   If the program runs successfully, you should see messages like the following:

   ```
   INFO mapreduce.Job:  map 100% reduce 100%
   INFO mapreduce.Job: Job job_local2110115835_0001 completed successfully
   ```

   followed by a set of statistics related to the job.
4. If the program succeeds, you'll see that MapReduce creates an output folder named `outputN`, where `N` is the number of the problem.

   To see the results, use the `cat` command on the output file, which is named `part-r-00000`:

```
cat outputN/part-r-00000
```

where you replace `N` with the number of the problem. (For at least one problem, the final results will be in a different folder named `outputN_final`. See the problems below for more detail.)

5. To remove the `*.class` files, the `*.jar` files, and the `output` directories, enter the following command:

```
make clean
```

Important guidelines

- Do **not** put your Java files in a package. Some IDEs like Eclipse will attempt to automatically add your files to a package, and you should prevent this from happening.
- Employ good programming style. Use appropriate indentation, select descriptive variable names, localize variables, insert blank lines between logical parts of your program, and add comments as necessary to explain what your code does.
- For each problem, you will need to define both a class for mapper tasks and a class for reducer tasks. As discussed in lecture, these classes should be static nested classes of your program class. See our `WordCount.java` file for another example of what the nesting should look like.
- Your nested classes should extend the built-in `Mapper` and `Reducer` classes, as discussed in lecture and shown in `WordCount.java`. Your mapper class will override the inherited `map()` method, and your reducer class will override the inherited `reduce()` method. You should *not* override or make use of the other inherited methods of the `Mapper` and `Reducer` classes.
- You are welcome to include additional helper methods in your classes as needed, although doing so is not required. You may also include additional helper classes for a given problem, but they should go in the same file as the rest of the code for that problem, and they should also be nested static classes.
- You should *not* use any global variables (i.e., static class variables) in your programs. Class constants (i.e., static *final* variables) are fine if needed.
- We have given you a `main` method in the starter code for each problem. You should make changes as needed to the class names in the method calls made within `main`.
- The `Makefile` passes in the appropriate input and output folder names as command-line arguments to your Java program. You will
need to ensure that your `main` method:

- uses the first command-line argument (`args[0]`) when calling `FileInputFormat.addInputPath()` for the job, as we do in the starter code
- uses the second command-line argument (`args[1]`) when calling `FileOutputFormat.setOutputPath()` for the job, as we do in the starter code.

Problem 3: Email domains

10 points

Write a MapReduce program to find the number of users for each email-address domain. The output of the program should be a list of (key, value) pairs in which the key is an email-address domain and the value is the number of users that have addresses from that domain. For example:

```
icloud.com   500
hotmail.com  250
...
```

(although you will probably get different numbers than the ones shown above!)

Notes:

- Your program should allow for large numbers (i.e., long integers) in the final results.
- Don't forget to specify the correct classes in the `main` method.
- If you use our `Makefile` to run the program, your results should end up in a folder named `output3`.

Problem 4: Email domain with the most users

15 points

Write a MapReduce program to find the email domain with the most users. The final output of the program should be a single (key, value) pair representing the email domain with the most users and the number of users that it has. If there are multiple email domains that are tied for the most users, your program may report any one of them.

Notes:

- You will need a chain of two MapReduce jobs for this problem. We discussed how to do this in lecture.
- The second job will process the results of the first job. Because those results are stored in a text file, the key-value pairs passed into the second `map` function will have the following format:

  - The key will be a file-offset value that you should ignore (just as any other `map` function that processes data stored in a text file should ignore its keys).
  - The value will be a `Text` value consisting of one line from the results file produced by the first job. In other words, it will contain one of the key-value pairs written by the first job's `reduce` function, with a single tab character (`"\t"`) between the key and the value.

- In the second job, the `map` function should ensure that all of the (key, value) pairs that it outputs have the same constant key. This will ensure that *all* of these pairs go to a single reducer task, which will then be able to determine which email domain has the largest number of users.
- Don't forget to specify the correct classes in the `main` method.
- Because your program will run two jobs, you should end up with two output folders: `output4`, which will hold the results of the first job, and `output4_final`, which will hold the results of the second job.

  Note that when configuring the *first* job, the `main` method uses `args[0]` for the input path and `args[1]` for the output path as usual. When configuring the *second* job, however, it uses `args[1]` for the input path (because the second job reads the results of the first job) and `args[2]` for the output path.

Problem 5: User with the most friends

15 points

Write a MapReduce program to find the user with the most friends in the social network. The final output of the program should be a single (key, value) pair representing the ID of the user with the most friends and the number of friends that this user has. If there are multiple users with the most friends, your program may report any one of them.

Notes:

- You should decide whether you will need a chain of two MapReduce jobs or whether a single job will suffice. The starter code includes a `main` method that configures a single job. If you need a second job, you should add the necessary lines to `main`, using the starter code for the previous problem as a model.
- If you use our `Makefile` to run the program, the final results should end up in either a folder named `output5` (if you use a single job) or a folder named `output5_final` (if you use a chain of two jobs).

Problem 6: Mutual friends

20 points

Write a MapReduce program to find the mutual friends between all pairs of friends in the social network. The output of the program should be (key, value) pairs in which the key is a pair of friends and the value is a list of other friends (if any) that those two friends have in common. For example:

```
1,2  3,4,5
1,3
```

Notes:

- The results should *not* include mutual friends of two users who are not themselves friends in the social network.
- You should only need a single MapReduce job.
- If you use our `Makefile` to run the program, your results should end up in a folder named `output6`.

Suggested Approach:

- Your mapper function should emit (key, value) pairs where the key represents a friendship between two users and the value represents the friends list of one of those users. For example, if user 1 has friends 2, 3, and 4, then for user 1 the mapper function should emit:

  ```
  1,2 -> 2,3,4
  1,3 -> 2,3,4
  1,4 -> 2,3,4
  ```

  Where the tokens on the left are the keys and the tokens on the right are the corresponding values.

  Later, when user 3 is processed (who has friends 1, 2, and 5), the mapper function should emit:

  ```
  1,3 -> 1,2,5
  2,3 -> 1,2,5
  3,5 -> 1,2,5
  ```

  Notice that the users in each key are listed in numerical order to ensure that the two friends lists for a given pair will be fed into the same reducing function. For example, we want `1,3 -> 2,3,4` (the friend list of user 1) and `1,3 -> 1,2,5` (the friend list of user 3) to be processed in the same invocation of reduce so that we can find the common friends in the two friends lists. For this to happen, the keys should be the same.

- We recommend using `Text` as the output type of the mapper's *key*, with each key being the *ordered* IDs of the two relevant users, separated by a single comma.
- For the output type of the mapper's *value*, one option is to use the `IntArrayWritable` class that we've provided as a static nested class in the starter code for `Problem6.java`. This

essentially allows you to represent the list of friend IDs as an array of integers.

An alternative option is to use another `Text` value consisting of the friend-list IDs separated by commas.

- Your reducer function should iterate over the two friends lists given for a specific key, compute the *intersection* of the friends in those lists, and output the list as `Text` in comma-separated format.
- You may find it helpful to use a built-in Java collection class like `HashSet` when determining the intersection.

Submitting your work for Part II

You should submit only the following four files:

- `Problem3.java`
- `Problem4.java`
- `Problem5.java`
- `Problem6.java`

Important: You *cannot* drag files stored inside the virtual machine to an application running outside of the virtual machine. Instead, you should submit everything from *within* the virtual machine by using the copy of Firefox that is installed on the VM.

Here are the steps:

1. Click on the icon for Firefox that is found on the desktop of the virtual machine.
2. Login to Gradescope by going to `www.gradescope.com` on Firefox.
3. Click on the box for ***CS 460***.
4. 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.)
5. 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.
6. Click the ***Upload*** button.
7. You should see a box saying that your submission was successful. Click the `(x)` button to close that box.
8. 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: For now, you will only see the results of a test that checks that the four required files were submitted, and that you didn't accidentally submit one of the original, unchanged starter files.

9. 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.**

10. 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`