# Lab 7: Intermediate Hadoop Programs

**Due date:** March 1, 11:59pm.

## Lab Assignment

### Assignment Preparation

This is a pair programming lab. You can pair with anyone in the class. I allow a small number of inter-section pairs. One team can consist of three students (to account the for number of students in the course).

### Overview

For this lab, you will implement a number of MapReduce jobs that run on the BeFuddled and ThghtShre datasets that you generated. For one of the MapReduce jobs you will conduct a performance evaluation, determining, at which point it is faster to run the job on Hadoop, rather than perform the same task using a straightforward sequential implementation on a single machine.

## BeFuddled Tasks

You will write three MapReduce tasks for the BeFuddled dataset.

**Input.** The input data for each of the three programs shall be a single JSON file containing the collection of JSON objects in the BeFuddled format. We will not worry about what your programs do if the input is different, although I recommend some form of graceful detection and exit. One thing to note is that the input file is a list of JSON objects, NOT a JSON array, like we did it in Lab 1[1].

[1] You can experiment. If you can make your MapReduce programs accept a JSON array and extract JSON objects from it one by one, feel free to keep the old format. Otherwise,

All of your programs shall accept two input parameters: the location of the input JSON file, and the location of the output directory.

## Program 1: Histogram of Moves

Create a MapReduce program `histogram.java`. The program shall produce a histogram of the frequency of different regular moves in the observed BeFuddled games.

**Output.** The expected format of one line of output is:

```
(x,y)   <frequency>
```

Here, `(x,y)` are the coordinates of the move (the `x` and `y` components of the `"location"` key in the JSON object), and `<frequency>` is the number of times the move to this particular location has been seen in all the observed games.

For example, if a move to position `(10, 13)` occurred 153 times, the output line specifying it shall look as follows:

```
(10, 13)  153
```

(Hint: create a `Point` class in your implementation, and override a `toString()` method to output the point coordinates in parentheses).

The coordinates of the point are the *key* of the output emitted by the `reduce()` method, while the frequency is the value emitted.

## Program 2: Game Summaries

Create a MapReduce program `summaries.java`. The program shall output a short summary for each game.

**Output.** The output of this MapReduce process is a collection of key-value pairs where the key is the unique Id of a game, and the value is a summary JSON object whose format is described below.

The summary object has the following format (I am using multi-line format for simplicity, in your output it can be rendered on a single line if needed):

```
"user" : <userId>,
"moves": <nMoves>,
"regular": <nRegularMoves>,
```

modify your Lab 1 generators to produce JSON objects without the JSON array syntax. The JSON objects can (and should be) multi-line.

```
 "special": <nSpecialMoves>,
 "outcome": <outcome>,
 "score": <finalScore>,
 "perMove": <avgScorePerMove>
}
```

Here:

| Value | Meaning |
|---|---|
| `<userId>` | Id of the user who played this game |
| `<nMoves>` | total number of observed moves in the game |
| `<nRegularMoves>` | number of regular moves in the game |
| `<nSpecialMoves>` | number of special moves in the game |
| `<outcome>` | `"win"`, `"loss"`, or `"in progress"` |
| `<finalScore>` | the score after the last observed move of the game |
| `<perMove>` | average number of points gained/lost per move in the game |

## Program 3: User Activity

Create a MapReduce program `activity.java`. The program shall output a short summary of each user's activity.

**Output.** The output of this MapReduce process is a collection of key-value pairs where the key is the unique UserId, and the value is a JSON object describing the user activity in the game.

The user activity JSON object has the following format:

```
{
 "games": <gamesPlayed>,
 "won": <gamesWon>,
 "lost": <gamesLost>,
 "highscore": <highestGameEndScore>,
 "longestGame": <largestNMoves>
}
```

Here:

| Value | Meaning |
|---|---|
| `<gamesPlayed>` | total number of games played (including any in-progress games) by the user |
| `<gamesWon>` | total number of games won by the user |
| `<gamesLost>` | total number of games lost by the user |
| `<highestGameEndScore>` | the highest score at the end of a completed (won or lost) game |
| `<largestNMoves>` | the largest number of moves in a signle completed (won or lost) game |

**Hint.** I recommend trying to get this job done by chaining multiple MapReduce jobs. Strictly speaking, it can be done in a single round, but it may be cleaner to complete in two or more rounds.

## 1 ThghtShre Tasks

You will write three MapReduce tasks for the ThghtShre dataset.

**Input.** The input data for each of the three programs shall be a single JSON file containing the collection of JSON objects in the ThghtShre format. We will not worry about what your programs do if the input is different, although I recommend some form of graceful detection and exit. One thing to note is that the input file is a list of JSON objects, NOT a JSON array, like we did it in Lab 1[2].

All of your programs shall accept two input parameters: the location of the input JSON file, and the location of the output directory.

### Program 4: Accounting

ThghtShre has decided to charge the users of the system for the communications (heh!). The charge model is as follows:

1. Each message incurs an origination charge of 5 cents.
2. Every 10 bytes (or any share of 10 bytes) of each message cost 1 cent.
3. If a message is longer than 100 bytes (characters), a surcharge of 5 cents is also assessed.
4. Users who write more than 100 messages get an overall 5% discount.

For example, the following message

```
Today is a wonderful day!
```

is 25 characters long. The cost of the message is the 5 cents origination charge plus 3 cents per byte charge, for a total of 8 cents.

Write a program `accounting.java` that computes for each user how much they owe the service.

**Output.** The output of this program is a collection of key-value pairs where the key is the user Id, and the value is the amount of money the user owes for their messages (in dollars and cents).

[2]You can experiment. If you can make your MapReduce programs accept a JSON array and extract JSON objects from it one by one, feel free to keep the old format. Otherwise, modify your Lab 1 generators to produce JSON objects without the JSON array syntax. The JSON objects can (and should be) multi-line.

## Program 5: Hashtagging.

`ThghtShre` decided to associate hashtags with each user account. The hashtags are the most popular words the users use in their messages, except for the words from the stopword list specified below. (If there is a single word most commonly used by a user, only one hashtag is assoicated with him/her. If there are two or more words used with exactly the same frequency, all these words form individual hashtags associated with the user).

The stopwords (i.e., words that do not count as potential hashtags) are:

```
a
the
in
on
I
he
she
it
there
is
```

Write a program `hashtags.java` that produces the hashtag assignment for each user of the service.

**Output.** The output of the program is a collection of key-value pairs, where the key is the unique Id of a user and a value is a comma-separated list of hashtags.

For example, if a user `u03243` has hashtags `"fast"` and `"fortune"` associated with them, then the appropriate output line shall look as follows:

```
u03243    fast, furious
```

**Hint.** This *may* require multiple MapReduce jobs chained together.

## Program 6: Popular words

Write a program `popular.java` that outputs the list of words used by the users in descending order of popularity.

**Output.** There is no preset format for the output here, but:

1. Each line of output must contain one word.
2. Each line of output must contain some means to verify the frequency of the word's use.

3. The words are outputed in descending order by popularity/frequency. The secondary sort order (i.e. for words with the same frequency) is left up to you.

## Performance Analysis.

For the performance analysis part of this lab, implement a sequential version of Program 1 and compare its performance on inputs of different size, to the performance of the MapReduce version.

Your goals are:

1. Capture the overall performance of both implementations as a function of the size of input (in bytes, in number of JSON objects).
2. Determine the performance trends for each of the two implementations (you can use your intuition here, or use regression).
3. Determine if there is an "inflection point" - the size of input at which it is more advantageous to use MapReduce for this task, than it is to use a straightforward implementation.

Create a short report (submit it as a PDF document) documenting the results of your experiment. The report shall briefly outline both of your implementations (include pseudo-code for map(), reduce() and the sequential implementation), describe your experiment, show all the obtained data in graphs and tables, and provide some analysis of the observed results.

To make it fair, run all your programs on `cslvm31` (or on any of the other machines of the Hadoop cluster).

## Submission

Submit your Java programs and the report.

All submitted files must contain your name on them.

Submit all your code in a single archive (zip or tar.gz).

Use `handin` to submit as follows:

Section 01:

```
$ handin dekhtyar lab07-01 <FILES>
```

Section 03:

```
$ handin dekhtyar lab07-03 <FILES>
```

**Good Luck!**