MAPREDUCE: PROGRAMMING CHALLENGES

MANAGING BIG DATA (MBD)
2017/2018

DOINA BUCUR

These MapReduce programming challenges in Python use only *small data*, so can be run on any computer. You can also work on a Farm cluster machine (which for the first two weeks you only use as a regular desktop computer, rather than a cluster node). Work in pairs if possible (but solo is also fine).

These challenges will get you used to thinking about processing data by sticking to the MapReduce basic type: *(key, value) pairs*, and by designing MapReduce parallel programs which have the right *complexity* so that they will work on *big data*, too, without overloading and crashing Hadoop cluster machines (I will specifically check this when testing your submissions!).

In Lecture 2, you have two MapReduce programs in pseudocode (computing a *word count*, and an *inverted index*), together with some small-data files. Make sure you understand this code first.

Writing programs over this simulator is a stepping stone before working with Spark, the MapReduce extension we will use in the week after this, on the Hadoop clusters.

See your Google Drive folder, subfolder `Programming challenges`, then `MapReduce-code`. You have there the two example programs from Lecture 2, now in Python, ready to execute (`word_count.py`, `inverted_index.py`).

Any program first imports the MapReduce simulator, `import MapReduce`. This simulator (implemented in `MapReduce.py`, and which you need to copy on your computer) is simpler than its Hadoop implementation:

(1) It executes Map and Reduce tasks sequentially.

(2) There's one input file. The simulator automatically creates as many Map tasks as lines are in the input file. You'll see that each line in each input file has the form `[key, value]`.

(3) It creates as many Reduce tasks as different keys are emitted by your Map tasks!

(4) The output of Reduce tasks are collected, sorted, and printed to standard output. (In practice, on a cluster, the output of a Reduce task is a file on disk.)

The simulator also prints in the output some complexity metrics for your program (see the lecture for a description of these metrics).

What to submit. Each Python MapReduce programming challenge has an identifier written in all-caps (e.g., MR-HASHTAGS). If you've solved a challenge, name the file: `MR-HASHTAGS-s123456-s234567-RND.py`, where the s-numbers are the student ids in the group, and `RND` is a short random string of your choice (this is needed to prevent others reading your file). *Important*: at the top of the file, add a comment block with: the full names and student numbers of the (pair of) students who wrote this submission. Don't forget this: it serves as a safety check alongside the s-numbers in the file name.

How to submit. We use a simple drop box on the Farm cluster. First, copy your code to your Farm machine. Then, in the folder where your code is, do `cp MR-HASHTAGS-s123456-s234567-RND.py /home/bucurd/MBD`. The latter folder is a write-only folder on the Unix NFS file system, accessible from any Farm machine; if the command exits without error, your code was copied. You don't have permissions to list (`ls`) that folder.

Points towards your final grade. Your grade from all programming challenges (MapReduce, Spark) will be "capped" at 2 grade points. Each challenge has grade points attached; for example, if that is *0.2 grade points*, then you can obtain up to 0.2 for your final grade by solving that challenge. No challenge is mandatory: solve as few or as many as you like! You may choose freely which challenges to attempt.

Plagiarism. In case you haven't studied at the UT before and don't know the rules: **never ever exchange any code**! You may talk about your solutions, but no line of code can be pasted between students.

(grade points: .1)

Counting words with better complexity (challenge identifier: **MR-WORDCOUNT**)

Add a Combiner to the `word_count` MapReduce implementation that you already have. This Combiner is essentially extra code in the Map tasks, which does some Reduce-like work already in Map tasks. It is up to you to figure out what exactly could be reduced in a Map task.

This addition will lower the amount of data in the output of Map tasks, which then means also a smaller input to Reduce tasks (so, less load on the network in a cluster environment, and lower Reducer size and skew).

Input

Read the local file `book_pages.json` in the input. This file stores book pages, one pair of the form `[title, page_text_as_a_string]` per line. A word is defined simply here as any non-white-space contiguous string.

Output

The output is a list of word counts across all the texts in the input:

```
["what", 5]
["wheel", 1]
["when", 2]
...
```

These word counts should be identical as in the case of the MapReduce program without a Combiner.

The complexity metrics printed in the output for you should be lower, particularly in the maximum Reducer size.

(grade points: .1)

Compute a distributed maximum (challenge identifier: **MR-MAX**)

Here's a simple task: compute the maximum of many integers distributedly.

Input

Read the file `integers.json` in the input. This stores one "batch" of integers per line, so a line has the form `[key, batch]`, where:

(1) the key is a numeric identifier for the batch, for example `0`

(2) the batch is a list of integers, for example `[199705, 213905, 212829, 13457, ...]`

Output

The program outputs a single number: the maximum value among all batches of integers.

Design the program so that it has the lowest complexity possible.

(grade points: .1)

Find frequent hashtags (challenge identifier: **MR-HASHTAGS**)

Given a number of (Dutch) tweets, calculate those hashtags which appear at least 20 times among these tweets. If a single tweet contains one hashtag multiple times, it's up to you: either count all the copies independently, or count only one of them.

Input

The input file `one_hour_of_tweets.json` stores one (key, value) pair per line, `[key, tweet]`, where:

(1) each `key` is a unique, numeric tweet identifier

(2) each `tweet` is a Python dictionary:
`{"retweeted":false, text="Text of tweet #here.", user="...", ...}` .

A hashtag can be defined simply here as any word prefixed by `#`. You can improve this definition if you like; for example, a hashtag could be any *part* of a word starting from `#`, e.g. `#here` in a tweet with typos: `"Test of tweet#here."`.

Output

Print in the output those hashtags which appear at least 20 times, and how many times each hashtag appears in tweets. This will look something like:

```
(hashtag1, count1)
(hashtag2, count2)
(hashtag3, count3)
...
```

Note that the MapReduce simulator sorts your output (which means: you'll see the list above sorted by hashtag).

(grade points: .3)

A distributed sort (challenge identifier: **MR-DSORT**)

Take many integers in the input; you are told that all integers are between 1 and 10^6. Sort them.

Input

The input file `integers.json` in the input. This stores one "batch" of integers per line, so a line has the form `[key, batch]`, where:

(1) the key is a numeric identifier for the batch, for example `0`

(2) the batch is a list of integers, for example `[199705, 213905, 212829, 13457, ...]`

Assume that one batch (line) in the input is *small data*, but the entire input is *big data*.

Output

It is up to you what format the sorted numbers are output in. The entire input should be globally sorted in the output; it should be a piece of cake for someone to pick up your output and verify that all the numbers come in the correct order.

Important for this problem: You should be able to argue why your number of Reduce tasks is a good choice!

There are a number of ways to solve this problem; choose based on feasibility in a big-data context.