MapReduce

Andey Robins Tuttle

1. Execution Instructions

For more complete execution information, see the `README.md` file in the root of the submitted zip file. Use the following command to build and run the classic example of a wordcounter. Expected output is shown after the command.

```
$ go run main.go -wc
Words = 20459
```

And my individual MapReduce problem, I took on the task of generating rainbow tables for exfiltrated passwords. The 10,000 most common passwords from this Github repository (`https://github.com/berandal666/Passwords/blob/master/10k_most_common.txt`) were used as the input. The output is a five column csv where the columns are plaintext password, md5 hash, sha1 hash, sha2 hash, and sha3 hash respectively.

```
$ go run main.go -rbow
```

Complete output is not shown as it is generated and placed in the file `rainbow.csv`.

2. Problem Statement

While securely storing passwords is a well understood problem, best practices are not always followed. Salting passwords and using a password sensitive hash function, such as bcrypt, are expected, but not always assured steps to properly store passwords. Many passwords are hashed with md5 or shaX algorithms and might not be salted before hashing. As a result, rainbow tables are an invaluable resource for password recovery. The problem tackled by this tool is generating hashes for a list of compromised passwords. An example attack scenario would be comparing recovered password hashes with the rainbow table to determine if any of the most common passwords are used in an exfiltrated password database.

3. Environment and Structure

I made use of Go and the Glow library (`https://github.com/chrislusf/glow`) as my MapReduce system. The software was built with Go version 1.19.1 darwin/arm64. Default file paths are generated in the Unix style instead of being platform agnostic. Providing explicit paths for other platforms is supported through command line arguments.

I made use of one Map and one Reduce function[1]. First, a Map function takes a password string and computes the md5, sha1, sha2, and sha3 unsalted hashes for this password. These hashes, as well as the plaintext password, are combined into a comma separated string which is returned from the Map function. After the Map, the Reduce function simply concatenates strings to generate the CSV file contents.

[1]The Glow library also uses the Map interface to define how keys are generated from the input file and how to output data. While these are two more Map functions, I don't count them in my functions because they don't perform interesting work. They only reads in strings and sends them to the "working" map functions and then write the final result to a file.

4. Results

For comparison of wall-clock time between MapReduce and more traditional, imperative programming approaches, an identical, imperative implementation of the work is provided. This can be run with the command:

```
$ go run main.go -imp
```

A single binary was built and executed with the associated flags (either `-rbow` for MapReduce or `-imp` for the imperative implementation). These were timed using the built in `time` utility on macOS Monterey 12.6.

```
$ go build
$ time ./mapreduce -rbow
./mapreduce -rbow  0.56s user 0.24s system 214% cpu 0.371 total
$ time ./mapreduce -imp
./mapreduce -imp  1.66s user 0.30s system 169% cpu 1.158 total
```

We can therefore see that the MapReduce approach to the problem ran more than 300% faster than the associated imperative solution. As a tradeoff though, it had an approximately 20% higher CPU utilization.

The generated rainbow table is available in the file `rainbow.csv`.