CMPE382

Operating Systems Project #3

Creating Dictionary for the AOL Dataset

Due: 11/06/2021

You are to do this project in groups of 2 persons.
This project must be implemented in C

There are four objectives to this assignment:

- To modify an existing code base.
- To use Threads along with disks.
- To understand and observe how disk works.
- To experience high performance computing and its intricacies in a real life problem.

Notes

- It is okey to use code or other material you find from outside sources (such as the Web), **given that you also provide proper citations.**
- When compiling and linking, you should use the argument **-pthread** to the compiler. This takes care of adding in the right libraries, etc., for using pthreads.
- It is OK to talk to others in class about what you are doing. It is even OK to explain to someone how some particular things works (e.g., how `pthread_create()` works). However, it is **NOT OK** to share code. If you are having trouble, **come talk to us instead.**

Background

As a part of this project, you are given a large Web Search query log which you will use throughout this project. Below, you can find the Dropbox link to the folder containing the query log:

https://www.dropbox.com/sh/i41bxgm5tou6r62/AAA0BBy4RKlpkhahNlAlTWvja?dl=0

Make sure that there are 10 files within this folder, named as Data1 to 10.txt. Each of these files contain exactly one query at each line. A query can contain, letters, numeric characters, and punctuation marks. You are not responsible for cleaning or processing the queries, however if your implementation allows you to execute a cleaner and more functional code, you are free to do so.

Note: The files are pretty large, and thus it will take some time to download. Make sure that you start downloading them as soon as possible.

Objective

The objective of the project is actually very simple. We ask you to build a dictionary for these query logs and write the dictionary into a file named "Dictionary.txt". In this file, each line will correspond to a unique query followed by the frequency of the query within the Data files. For example: If query "Winter Gardening" appears 20 times within all 10 files, the line representing this particular query will be "Winter Gardening\t20" (the query and the frequency should be separated with a TAB character, i.e., \t) (Note that terms within a query are separated with spaces and frequency is separated from the rest with a tab, making the processing easier)

During the project we will examine different strategies and compare them with each other. In order to ease the tasks at hand, the project is divided into several tasks:

Task 1: Building a Memory-Based Data Structure for Building the Dictionary

Basically, what you will do is read each of the files line by line and build a data structure that will keep track of the queries (whether they appear previously or not, and if so, what was their frequencies).

A suitable data structure for this task is called a "TRIE" data structure. You make refer to any outside source to find out how a trie works and how it can be implemented.

As the first task of your project, implement a Trie data structure. You are also allowed to use an already existing implementation from outside sources. However be warned: common implementation of a trie usually is not implemented to keep track of frequencies. Also most tries do not count the space character as a valid character. Thus, you may need to do modifications even if you can find an already implemented trie. Possible methods are:

- Implement the trie yourself.
- Modify a trie implementation so that it suits your goals.
- Keep additional data structures to trie so that you can keep query frequencies.

Task 2: Sequential Execution

The pseudocode for your program will be something like this:

```
1. starting from the first file, read one query (that is one line) from the Data File
2. Insert it into the Trie.
    a. If already in the Trie, increment its count
    b. If not in the Trie, create a new branch and set its count to 1.
3. When files are finished write the resulting Trie into the output file.
```

You should observe the execution time of this program when you are finished.

Task 3 : Sequential Execution - Multiple Queries

Use a very similar execution with the first one:

1. starting from the first file, read a block of queries (such as 1024 bytes, 2048 bytes and so forth) from the Data File into a data structure in the memory. Note that you have to parse the string to get queries in this scenario to get queries one by one. Difference from the first scenario is that now you are performing a single read operation for multiple queries instead of performing one.
2. For each item you read, Insert them one by one into the *Trie*.
   a. If already in *Trie*, increment its count
   b. If not in *Trie*, create a new branch and set its count to 1.
3. When files are finished write the resulting *Trie* into the output file.

As you can imagine, the output will be exactly the same. However we are testing whether reading a single item from disk or multiple items benefit our code or not (Maybe it was already doing it in Task2 automatically?)

Try to go well over 16, like several thousands, and draw a plot of the execution times to see its effects.

Task 4: Threaded Execution

In this task, Task 2 is repeated with a very small modification: Each file is assigned to a thread and processed this way. Each of the threads perform Task 2 to their local files, and use a single trie to write the queries.

(The problem you will immediately observe will be since all threads are accessing to a shared data structure, the execution will be impaired, and the results may become inconsistent) First observe what will go wrong, then define a mutex that will prevent multiple threads accessing to the data structure at the same time.

Observe execution times and compare with previous methods.

Task 5: Threaded Execution - Multiple Tries

In this task, Task 4 is modified slightly. Each thread instead of using a shared trie, first create a local trie, then when all threads are finished, these tries are merged in a sequential fashion (The naive strategy is read queries from each trie one by one and create a final global trie. Then use this trie to generate the dictionary file.)

In case your computer's memory is not sufficient to keep all of the tries in memory (which is possible) you might think of first writing local dictionaries to the disk as a file, then perform a dictionary merge.

Task 6: Completely Memory-Based Dictionary Creation

Your task is to implement yet another modification to Tasks 2 and 3: Instead of reading queries one by one or in small groups read all queries into memory first. (Do it sequentially, you can anticipate what will happen if you use threads at this point thus asking you to implement that version will be redundant.) Then start dictionary creation using the data in the memory, without using any disk accesses.

Observe its execution time and as always, compare it with other methods.

Task 7: Improvement

In this task you are asked to find a method that **may** improve the execution time of one of the above strategies and implement it. Every type of improvement will be accepted, even the simple and stupid looking ones. (Though you will be asked to defend your improvement either theoretically or via implementation (very unlikely).)

Grading

Hand in your source code and a minimum 3, maximum 6 page report showing your results and explaining your improvement. Note that you code does not count for fulfilling the page count, we want discussion.

You may want to submit multiple source codes, if so please also prepare a README file explaining which code/executable is intended for which task. Your readme should also explain how your programs are executed.

In your `README` file you should have the following five sections:

- The name and login information .
- Design overview: A few simple paragraphs describing the overall structure of your code and any important structures.
- Complete specification: Describe how you handled any ambiguities in the specification.
- Known bugs or problems: A list of any features that you did not implement or that you know are not working correctly

Important Final Note: Note that we already know the provided files are quite large and you may have rough time implementing each part if you are using an old computer. Although not ideal, you can shorten the files, dismiss some parts of them while you are working. That would be much better than not submitting anything.

Even more important final note: If you do modifications on the files, dont forget to mention those in your report.

Last important note: Files contain queries at each line, and each query is composed of terms. A dictionary contains terms, not queries.

Good Luck and Have Fun!