COL362/632: Introduction to DBMS

Assignment 3

Due: 15 April 2023 11:59PM (**hard deadline, will not be extended**)

Version History

Revision	Date	Author(s)	Description
1.0	04.04.2023	TAs	Initial Version.

GENERAL INSTRUCTIONS

1. Please complete the assignment *in pairs*. No collaborations with other entities – person(s)[other than your partner], AI agents, websites, discussion fora etc.– is allowed.
2. **Submissions have to be made by both members of the team**. If only one member submits, we will ignore the submission.
3. **Marking scheme:**
   (a) If the program fails to successfully terminate within a reasonable time for a given workload, we will kill the program and award 0 marks. What constitutes a *reasonable* time will be dependent on the workload size and will be determined by the instructor.
   (b) Correctness of the implementation - 25%
   (c) If the implementation produces correct final output, for producing correct intermediate output as per the arguments given - 25%
   (d) If both above steps are correct, the efficiency of implementation will evaluated against 40% of assignment marks as follows - all submissions will be sorted in decreasing order based on their average running time over 2 different workloads (averaged over 3 runs for each). Full marks for fastest-5% submissions, and progressively reduced marks in steps of 5 percentile as we go down the sorted order of performance.
   (e) Submitted report containing all the necessary details - 10%
4. **Submissions:** You will submit *two files*:
   (a) a pdf document named <entrynumber1_entrynumber2>.pdf and
   (b) *one* C/C++ source file <entrynumber1_entrynumber2>.c (or cpp/C).

No submissions in Java, Python or any other language will be accepted. It is necessary for your C code to build and compile using Gnu-C compiler version 5 on a Linux machine. We will *not be able to build your code on Windows/MacOS or any other platform.*

5. Your submissions will be subjected to strict plagiarism checks.

PLAGIARISM RULES

If caught, you will receive 0 for the assignment, and a **penalty** of −10% in the course.

1 Efficient External K-way Merge-sorting

In this assignment, the goal is to implement a fast K-way merge-sorting of a large number of *character strings* (for simplicity we will assume all strings have only ASCII characters). You are allowed to use any tricks for speeding up as long as you adhere to the constraints imposed by the arguments. For details see the detailed description below:

1.1 Workload

We use the term *workload* to refer to a disk-based file consisting of variable length strings consisting of ASCII (non-control) characters (you can assume them to be less than 1024-bytes). Each string is terminated by a newline. We will use two different workloads – first, which contains English language words/phrases; and the second, which contains purely random sequence of characters. The first workload is called `english.list` and `random.list` – you must use the same terminology in your submissions as well. Also note that you can not make any assumptions about the key distribution in these files – there could be duplicates, there could be strings that are very small, there could be strings that contain special characters (such as !, #, etc.), and so on.

The size of these workloads will be large (at least 10-20 times the available memory). Smaller scale datasets (or the script to generate them) will be released *two days* before the submission deadline. This is to help you to test your code against specific kind of dataset that will be used in the evaluation. For scalability and efficiency, you may have to create and use your own datasets.

1.2 Code structure and arguments

All your implementation must be in a single file (no additional header/source files). You will be required to implement the following function which will be called from a main program that sets up the argument values. **You must not change the function name or the argument order**.

```
/*
 * The function is expected to implement a k-way external merge-sort
 * on a file consisting of string values. The description of arguments
 * is given in the assignment document.
 */
int external_merge_sort_withstop (const char* input, const char *output,
   const long key_count, const int k = 2, const int num_merges = 0);
```

The description of each argument is given in Table 2. All intermediate runs will be written to files named as `temp.<stage>.<runnum>`. For instance, when you create the first sorted runs from the input file, you will output `temp.0.1`,`temp.0.2`,... and so on. The output after merging these two runs will be `temp.1.1`,`temp.1.2`,.... etc. It is important for you to follow this convention strictly since we will use this to monitor if your submissions are correct.

Note that you are free to use any strategy to speed your program. For instance, you can use any data-structure / algorithm to implement the in-memory sorting step, you can use any form of compression to reduce the intermediate disk I/O. However, if you use anything that is not part of the standard libraries available on a Linux machine, you are expected to include the source-code in the submission file. For example, if you want to use trie-structure for in-memory sorting of strings, you will have to implement it yourself and include it in the submission file since it is not available as part of any standard Linux library. **Only the input and output files will be in ASCII format (i.e., they should be human readable strings).**

Parameter	Description
input	Full path of the file containing input strings
output	Full path of the file where the sorted output should be written
key_count	Number of keys in the input file that need to sorted. Note that there may be more keys in the file, in which case you will sort only the **first key_count keys** from the file
k	the maximum fanout of the merges that are allowed. Note that this does not limit the number of runs or the size of each run that you are allowed to create. It simply restricts the number of runs that can be merged at once. Default value = 2.
num_merges	This is an optional argument that is given to stop the sorting process midway and inspect the intermediate results. If not specified (or set as 0), the function continues until the final sorted output is produced. Otherwise, it will stop after performing **num_merges** number of merge steps. A merge-step is defined as the reading k runs, and writing out a merged output to a file.
return value	On successful completion, the function returns the number of merge-steps completed (see above). It returns a value less than 0 if it failed to complete successfully.

Table 2: Description of each parameter to the function `external_merge_sort_withstop`

1.3 Evaluation Platform

All programs will be built and executed on a Linux machine with single CPU, and 1GB of RAM (we will use -O3 compiler optimization flag, with no debug symbols enabled). You can assume that there is sufficient hard-disk space to hold the intermediate runs and the final sorted output file (apart from the input dataset). No additional disk space can be used.

1.4 Evaluation

Our focus in this assignment is optimizing the wall-clock time used to complete the sorting in a single-threaded program. The main program that will call your function will count the number of milliseconds elapsed between the start and end of the function call.

For each workload, we will run your program 3 times with 3 different values of k ($= 2, 8, 16$), and the average of all runs will be used as the measure of the program performance. The total ranking will be based on the sum of the performance numbers for each workload.

For verifying if the program is doing what is expected, we will vary the k and num_merges arguments, and check if the program is behaving as required. These are not used in wall-clock measurements.

1.5 Submissions

You must submit two separate files, one file containing the implementation in C/C++ and the other file consisting of a report about your implementation as well as the performance measurements that you have done. In detail:

Report should contain the details of the design choices made in your implementation such as data-structures used for in-memory sorting of strings, strategies used for optimizing the disk I/O, the datasets and methods used for performance evaluation, what was the observed performance over the dataset(s) that was released, etc.

Source code of the implementation This is, as mentioned above, is a single C/C++ file that contains the implementation of the function given above. Note that we will have a evaluation harness code (main method) that will call this function, so it is important that you do not change it.

The naming convention to follow for these files is given in the instructions at the beginning of this document.