Aaron Kau, Ben Yang, and Kelvin Liu

README.pdf

This program consists of a client and a server. The client parses all CSV files in a directory and its subdirectories and sends these files, along with a label to sort by, to the server which sorts the rows in the CSV and merges them together into one large CSV file. The directory to sort and the output directory can also be specified to the client. The program makes use of various system calls, functions, data structures to make the sorting process quick and efficient.

Usage

To run the code, first compile by running the *all* script on the supplied makefile.

1. *make*

The server can be run by executing the following command.

2. ./sorter_server –p [port]

The client can be run by executing the following command.

3. ./sorter_client –c [column to sort by] –h [hostname] –p [port] –d [directory] –o [output directory]

where [directory] is the directory with CSVs to sort, [output directory] is the desired location of the sorted CSV files, [column to sort by] is the column name to sort by, [port] is the port to listen and connect to, and [hostname] is the hostname for the server. The -d and -o flags are optional. If they are not specified then -d and -o is assumed to be the current directory. Specifying the -o flag outputs the merged CSV file to that directory. The CSV files are checked to have the specified 28 columns; if not, then they are skipped (note that the thread is still created).

As an example, the *runs* script (called by running *make runs*) in the make file will host the server on port 3030, and the *runc* script (called by running *make runc*) will attempt to connect to the server (assuming it is hosted in man.cs.rutgers.edu port 3030) and have it sort directory “t” (provided as well) and output to directory “output”. It will sort and merge the CSV files in folder “t”, including movie_metadata.csv file (assuming it is in the same directory as sorter) by the column “director_name”, and output the sorted and merged CSV in a folder called output.

Assumptions

This program assumes that every file that is to be sorted has a name ending with .csv, and every subdirectory needs to have all its .csv files sorted as well. Furthermore, there should not be any other filetypes besides basic files and folders, and all filenames are unique (an issue if all sorted files are output to the same directory and have the same name).

This program also assumes that the header files (sorter_server.h, sorter_client.h) will contain the correct info about the CSV file's columns. The stringValues, intValues, and doubleValues arrays should contain the names of the columns that contain strings, integers, and doubles respectively (and stringValuesSize, intValuesSize, and doubleValuesSize should contain the length of these arrays).

The program will also initially allocate 5,000 bytes of space for each string and space for 10,000 rows in the CSV file (although it will automatically increase these global variables if they are exceeded).

The program will also assume the following about any null entries:

If the column stores strings, it is the empty string "".

If the column stores integers or doubles, it is the value zero (0 or 0.0).

Design

The client parses through each of the folders and directory within the input folder and for each file it finds, it creates a socket and connects to the server, sending a request for the file to be sorted. After going through every file and making the request, it makes a dump request to the server, where it requests a merged version of all the sorted CSV files.

The server uses a socket to start listening at the inputted port number. For every request that it receives, it creates a new thread that will accept the connection, read in the data (if it's a sort request and contains a CSV file, it will read in the contents of the CSV file and parse it as well), and respond with an acknowledgement of the request. If the request is for a CSV dump, it will merge all the sorted files it has collected and send it to the client making the request.

To parse the CSV file, we created several structs that contain information about the file including the data stored, data types for each of the columns, column names, as well as the number of entries being stored.

An enum called *type* was used for each of the possible data types that could be stored in the CSV (string, number, and decimal). The types for each of the columns in the CSV file were set beforehand using the F.A.Q. from the CSV file, although a CSV file with other column names and data types could be parsed by modifying the appropriate constants in the Sorter.h file (stringValues, intValues, doubleValues, and columns).

Each row in the CSV file was stored as an array of values. Since values could have differing data types depending on the column it resides in, value is saved as a union, which could be either an integer value, a double value, or a pointer to a string. The corresponding data type is retrieved from csv->columnTypes[columnNumber] to determine the type of that value.

Since multiple columns can be used to sort the entries, the query, for example [gross,movie_director], would be converted into an int array which holds the position of the headings, for example [8,2]. Whenever a comparison between two entries is done, the int array is used to compare the values at the indices specified by the array—the 8^{th} position, and if the values are the same, then the 2^{nd} position.

Mergesort

A mergesort algorithm was used to sort the CSV file by the specified column. The array of entries was split in half and the mergesort algorithm was recursively called on these subarrays. After these subarrays were sorted, they were merged together to form the fully sorted array using the algorithm to merge 2 sorted lists (Traverse through both lists with 2 pointers and keep adding the smallest value). Mergesort has a run-time of O(n*log(n)) with n inputs, where n is the number of rows for the CSV file.

Testing Procedures

Our program was tested by sorting various different combinations of csv files in various different number of subdirectories with varying column queries. We made sure to sort by each data type and we manually checked the output to ensure that in general, the resultant rows appeared to be in the correct order.

We also ran the program on directories already sorted to make sure that the CSV parser could take its own output as input and sort by another set of columns.

To check for CSV files of varying sizes, we also cut out varying number of rows to from the list to make sure that the CSV file could parse smaller CSV files, and we also appended the CSV file to itself (doubling the size of the CSV file) to see if the program could parse larger files.

Lastly, we changed several names of columns (and reflected such changes in our header file), and even changed the data types of many integer and double types to string and made sure the program was outputting correct CSV files based on these new column names and data types.

Challenges

Working with networks and sockets were particularly challenging because it was difficult to find a good way to serialize and deserialize the parsed CSV files on both the server and the client. We initially tried to output the CSV file to a file, and pass the contents of the file as a string to the server, but we found that special characters were messing with the socket connection. We ultimately ended up passing the request type, and size of the CSV before passing in any data so it knew how much to read at a time.

Another challenge that we faced was that the “read” command would sometimes not read the requested number of bytes (usually because the “write” command on the other end didn’t finish writing all the data). Even though read blocks until there is data to be read, it will read as soon as something becomes available, even if it isn’t enough to fill the buffer. This was especially difficult to detect because it would occur randomly and would happen in different sections of the code every time. We ultimately addressed this by adding a “force read” function that would check how many bytes were actually read in a “read”, and would loop and read again for more bytes until the buffer was completely filled up.