Script 2 Report

For the second part of the project, the same parsing system as the first part was used, which has the same advantages and disadvantages.

There are three catalogs created to keep the data of the respective files user.csv, repos.csv, and commits.csv. In detail, each catalog has an array of structs of the specific type (USER, REPOS, or COMMITS), the length of that array (int), and one hash table, except the commits catalog, which doesn't need an hash table. The hash tables have the id of the specific type as the key and the struct as the value.

Hash Tables aren't the most memory efficient and they take a long time to allocate memory and setting up values. However, they are incredibly fast at looking up values, once set up. As the second script requires a higher number of lookups, the team chose to use hash tables from the Glib library.

Now that the catalogs are ready, the file containing the commands is opened and read. The program separates the first word out of the string, which will be a number between 1 and 10. This number indicates what type of query this string is referring to, and, subsequently, the number of parameters that the program needs to parse. For each of the ten types of queries, a function was created addressing it, which are separated between statistical queries and parameter queries, for better organization.

Query type 1 counts the number of bots, organizations, and users. This is done quickly and efficiently because of how the catalogs were created.

The second query type calculates the average of collaborators per repository. Firstly, the program creates one hash that has as elements Garrays, one for each repository. For each commit, the User ID is put on the respective GArray. After all commits have been analyzed, the repeats are removed. With everything set, the program calculates the average. This method is fast, but memory intensive.

The third query type counts how many repositories have bots as collaborators. For each commit, the User ID is extracted, and the associated profile is looked up. If the user is a bot, the repository is added to a hash table. The hash table does not add duplicates. The function then prints the hash table length.

Query type number 4 divides the length of the commits catalog and the length of the users' catalog, using the included function with Glib.

The fifth query type calculates who the top N users with more commits in a certain date span. A struct was created to assist this process, called DATA, that creates dates and whose auxiliary function allows us to compare them. Commits are analyzed one by one, checking if they were created within the correct dates. If the commit is valid in that aspect, the users' counter is incremented. The counters are stored in a hash table, whose keys are the users IDs. After all commits are analyzed, the counters are inserted in a GArray and sorted.

Query type 6 is almost the same as the previous query type, but instead of checking a date, it compares the commit language with the function parameter language.

The seventh query type is incredibly like the fifth query type. The number of commits pointed at a repository are saved in a hash table exactly like the fifth query type. The function then prints all repositories that are not present in the hash table.

The eighth query type is a modification of the fifth query type, but instead of directing a date valid commit to its user, it checks the repository's language.

Query type number 9 mixes the processes of query type five and six. When a commit is analyzed, the user and the repository are fetched. From the Repository, the owner's ID is looked up and the friends array from the user is searched. If the repository owner ID is found in the array, the commit is valid. After that, it's the same process. The hash table counters are copied to a GArray, which is ordered, and the top N elements are printed.

Finally the last query type, query type 10. A hash tables is created for the repositories. When a commit is read, it is inserted into the hash table's respective value. After all commits have been inserted in their respective hash table value, the strings are sorted by length, and then printed.

Various auxiliary functions were used (included in the queriesAux moule). This file includes the auxiliary functions and modules that needed to be created to add a simpler option to certain problems. Not creating these would make the code slower and much more complex for someone to analyze.