CMPUT 291 Mini Project 2

Emmett Postill
Andrew Simon
Juan Diego Castro

Overview

This application reads the 4 records provided in eclass in XML format and processes them in two stages. First it produces an .txt file with the style specified in the assignment spec and then uses that to make 4 .idx indices that will be used to make queries on the data. Finally in the last stage it prompts the user to make a query following the provided grammar and to choose between seeing the full output or just the ids of the emails in the result (default mode)

User Guide

Phase 1:

The user will be prompted for the name of the input xml, which will then be used to create the .txt files needed for phase 2

Phase 2:

This requires no user input and only relies on phase 1 being ran first.

Phase 3:

The user will be asked to press enter to proceed q to quit or type "output=full" to change the default mode**.** The default output mode is brief (only the email ids of the result of the query will be returned). Then it will prompt to write a query that should follow the grammar provided in eclass

Software Description

Phase 1:

Phase 1 takes the input xml and generates 4 text files to then be converted into idx files in phase 2. The main function goes through the xml one line at a time, checking if there are lines with the "mail" tag. If there are, it goes through the line extracting all the relevant info from tags.

This is done using the parse_tag function. Parse_tag uses the regular expressions library to match lines with the tag, then strips the tag.

Once the info from the tags is collected, it is formatted into the text files, then saved

Phase 2:

Phase 2 is simply taking the txt files from phase 1 and using the linux sort commands to save them as idx indices (either binary tree or hash).

Phase 3:

The query system was designed with two principal functions (get_user_input() and query_system()) that call auxiliary functions to resolve the different types of queries.

Firstly the get_user_input() takes whatever condition the user is imposing on the data and breaks down it's input in atomic queries that are put each as elements of a list. For instance if the user types: " to: peter@ualberta.c confidential from   : john@ualberta.c   " this function returns [to:pete@ualberta.c,condifdential,from:john@ualberta.c]

This output plus the output format mode is then pass to query_system([list of atomic queries],output_mode) that will take each of the elements of the list and classify them into 4 query categories (body,subject,date and email address queries) according to the keywords they contain.

Following after the program will call auxiliary functions that will resolve the most efficient way to retrieve the result for the particular type of atomic query they are solving. The specifics of the functions that do this are found below.

Finally the output is then put into a list of list (one entry per each atomic query) and the intersection is taken as the email ids to be returned or hashed into the full email and retrieved if the output format is "full".

This means that for example if we make a query for : "to:castromi@ualberta.c subj:cmpt291" then to achieve efficiency the program would retrieve the emails ids for the individual separate queries using the appropriated tailored function for the two types of queries and then take the set intersection of the Ids provided by them.

In the event multiple query conditions are supplied, ie, body:stock confidential shares date<2001/04/12, the query_systems function will query for each query as though they are unique queries and then use the intersect function to ensure each row id is unique and that the only row ids that are used to query the subject and/or body are those that will be printed, which is more efficient.

Individual Functions

email_adress_queries(element from get_user_input output):

This function deals with the queries containing 'to','cc','from', and 'to'. It splits the input using the ':' string as marker, takes the first element and locates the first key related to it in the

Db_tree and adds the value element to a list. Then proceeds to iterates over all the entries with the same key using the curs.next_dup() until it hits a None element

wild_card_query(word):

This function queries for the partial match input. It takes a single word as an argument, and searches through the te.idx file. Since this searches only through the te.idx file to find partial matches rather than through the re.idx file, this is a more efficient approach.

fetch_output_info(rowID_list, output_mode):

This function returns a list of lists, where each list consists of either the row id and subject of the email, or the row id, subject, and body of the email, depending on the output mode. It takes a list of row ids and the current output mode as arguments. It searches through the re.idx file. This is the only time the re.idx file is searched through in the code, so the minimal amount of time is spent searching through re.idx, which is an efficient way of searching for the information.

print_output_info(output_info, output_mode):

This function prints the output from fetch_output_info, and changes what it prints based off of the output mode. It takes the output from fetch_output_info and the output mode as it's arguments.

intersect(IDList):

This function intersects the results from all the queries that have been performed. As an argument, it takes a list of lists where each list is comprised of the results of a query. So, if used on the list of row ids before they're passed to the fetch_output_info function, this function ensures that every row id passed to the fetch_output_info function is unique, thus the least amount of searching will be performed.

query_dates(day, oper):

This function queries for dates with respect to the operation versus the inputted date. It takes the date to be searched for and the operation by which to compare dates as the arguments. This function searches through the da.idx files to find the dates, so this is the most efficient approach.

query_subjects(word):

This function queries for a given word in the subjects of the emails. It uses the te.idx file. It takes the word to be searched for as it's argument. It searches through te.idx rather than re.idx, so it minimizes the number of characters it will have to search through.

query_bodies(word):

This function queries for a given word in the bodies of the emails. It uses the te.idx files. It takes the word to be searched for as it's argument. It searches through te.idx rather than re.idx, so it minimizes the number of characters it will have to search through.

Testing Strategy

Phase 1 testing was simply running the function on the given xml files and comparing the result files with the expected result files

Phase 2 testing was simply running the commands and checking to see that the data was properly sorted and queries in phase 3 run properly using them.

Phase 3 testing was checking correctness running queries against the .idx files created in phase 2. Multiple condition, empty, equality and range queries were covered

Group Work Strategy

Our group work strategy was dividing the work into independent roles, spending the last few days fully integrating each others work where needed.

Emmett was responsible for phases 1 and 2. He spent roughly 10 hours on the project

Juan Diego was responsible for writing the get_input() function that atomizes the queries, for the queries involving from, cc, to,bcc,cc keywords and for designing the draft for the architecture of how the phase3.py would work. He spent roughly 10 hours on the project.

Andrew was responsible for the queries involving subjects, bodies, and partial matching. He was also responsible for the functions that dictate the printing of the results of the queries. He spent about 10 hours on the project.