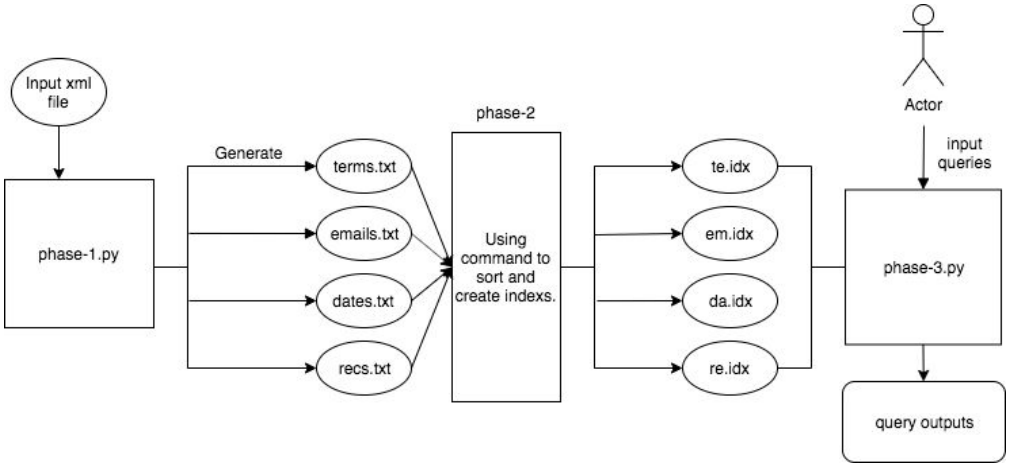
General Overview

The main idea of this program is to use Berkeley DB working with data in physical layer. There are 3 parts of the program. The phase-1.py will output terms.txt, emails.txt, dates.txt and recs.txt based on an input XML file. For phase 2 there is a bin file that execute sorting breaking and creating index for 4 text files, and phase-3.py is the program that handles all the input queries.



User guides

For phase 1: run the phase-1.py on terminal or using command “python3 phase-1.py 1k.xml(or put the name of whatever the xml file you want to use)”. This program will generate 4 text files: terms.txt, emails.txt, dates.txt and recs.txt

For phase 2, we use a bin file to sort, break and create index files. Using “chmod -x bin” to initialize first if necessary, and use “./bin” to execute.

For phase 3: The default of the output of each query is the row id and subject fields of all matching emails. If user input “output=full”, the format of the outputs will be full records and using “output=brief”, the output will go back to the format of row id and subject fields. For input queries, the format of inputs should strictly follow the formats on eclass “https://eclass.srv.ualberta.ca/mod/page/view.php?id=3659785”.

Algorithm description

For phase 2, we sort text files and create index files manually. (1) Sort recs.txt using “`sort -nu recs.txt -o recs.txt`” and then use perl to break down sorted file “`perl break.pl < recs.txt > recs_formatted.txt`”. Then use “`db_load -T -c duplicates=1 -f recs_formatted.txt -t hash re.idx`” to create hash index for recs.txt. (2) Use “`sort -u terms.txt -o terms.txt`” then “`perl break.pl < terms.txt > terms_formatted.txt`” to break and “`db_load -T -c duplicates=1 -f terms_formatted.txt -t btree te.idx`” to create b+tree index for terms.txt (3) Use “`sort -u emails.txt -o emails.txt`” and “`break.pl <`

emails.txt > emails_formatted.txt" then "db_load -T -c duplicates=1 -f emails_formatted.txt -t btree em.idx" to create index file. (4) Using command: "sort -u dates.txt -o dates.txt" to sort dates.txt, then use perl to break down sorted file: "perl break.pl < dates.txt > dates_formatted.txt". After that, we use "db_load -T -c duplicates=1 -f dates_formatted.txt -t btree da.idx" to create b+tree index file for dates.txt.

For phase 3: The "main" function starts the program and taking user input queries. If there are many spaces within a query, the "break_user_input" function will extract query only, and pass query to the "checkvalidity" function, which is the function that check if each query is syntactically correct. If not, the program will show the query is invalid and require another input. Moreover, it will check each query type. For example, if the input query is "subj:gas", it is searching in terms.txt, so the type of input query is "termquery". It also check that user wants full or brief output. The "process_query" function returns the prefix of query and what we want to look for in particular. "query_test" will use the returned prefix to find which idx file to open and select all data that matches. Then it will open recs.idx to match records based on matched data's row id to a dictionary. Query test returns all the results of one query, which is appended to a list called "all_query_result". After all queries outputs get into that list, we use "multi_query", which finds intersect row ids of all queries output lists. Then, the results are printed.
For range search, we use for loops to compare the inputted date with the iterated date using date time module. And for wild cards, we use re model to search and find if the partial in the index file.
We use dictionaries to get the row ids from each of the files. And we find the same row ids in all queries outputs and use re.idx to get specific output. For single query, the time is efficient. For multiple queries, since we iterate all outputs to find intersection, it takes time to generate.

Testing Strategy

discusses your general strategy for testing, with the scenarios being tested and the coverage of your test cases

For phase 1, we have used 10.xml and 1k.xml to test and compare output text files. However, there is no sample outputs for 10k.xml and 100k.xml, so we can only tested the generation text file function.
For phase 2, after generate index files, we used db_dump to visualize index and check it. For instance, for te.idx, use: "db_dump -p te.idx > te.txt". We manually check if it is correct. Coverage is 10.xml outputs and some of 1k.xml outputs.
For phase 3, we use output index files of 10.xml and 1k.xml. We checked if the output formats are correct or not, then search input conditions in the text file that from index using db_dump to check if all matching records covered.

Break-down Strategy

Nasif Hossain (ccid: nhossain) : Implementation and organize codes on phase 1. Fixed errors on emails.txt. Organize commands on phase 2. Break down and reorganize codes on phase 3. Time spent: 20 hours
Yazan Al-Muhtaseb (ccid: yalmuhta): Wrote codes for phase 1 for emails, dates, recs. Find sort commands and index commands on phase 2, finished 90% codes of phase 3 by himself. Time spent: 30 hours
Pengcheng Yan (ccid: py): Fixed errors on terms.txt, testing codes for phase 1 and report. Time spent: 13 hours
We use GitHub and Google Drive to keep project on track
Assumptions: Input query formats in phase 3 follow the formats on eclass. All input queries are reasonable.