CMPUT 291 Mini-Project 2 Report

Submitted by: Bardia Samimi, Kamillah Hasham, Neel Kumar

A) General overview and user guide:

main.py user guide

This file builds the `.idx` files from a user inputted `.xml` file

1. Run main.py
2. Enter the name of your `.xml` file
3. The text files, db_load text files, index files, and human-readable index files will all be created and outputted into the current working directory

phase3.py user guide

This program retrieves information from the `.idx` files

1. Run phase3.py
2. By default sets the database to "brief" mode, which prints matching row_ids and subject fields. This can manually be changed by typing 'output=full' and 'output=brief'.
3. Queries can be written in the format:
    a. [keyword] [operator] [term]
        i. Keywords: ["row", "date", "from", "to", "subject", "subj", "cc", "bcc", "body"]
        ii. Operators: ["<",">",":", ">=", "<="]
        iii. A term of the users choice
    b. Queries may be combined. Unnecessary whitespace will be omitted, for example:
        i. body:stock to :   davood@ualberta.ca confidential shares date<2001/04/12 foo foo%
4. Type 'EXIT' to quit.

query_creation.py user guide

1. Phase3.py calls query_creation.py to the program stack relaying user-inputted data as the parameter.
2. Dictionary output format is as follows:
   { "cursor name" : [operator + index key1, operator + index key2, . . .] }
3. To search the database, iterate through the dictionary cursor names and pull each associated value. The first char will be an operator and last char will be "%" to indicate partial matches. Remove these enclosing fields to get the index searchable key.

B) The detailed design of your software

Query Creation

- User inputted data is taken as a parameter and split into a character array.
- The character array is iterated through to find all symbols. Once found, enclosing whitespace is inserted.
- Array is rejoined into a new string where all operators and search terms are separated.
- This new string is split on whitespace to ensure that every entry is either an operator or search term.
- The new array is iterated through to find all operators, and using the word before them as keywords and the word after as values. Using this data, the dictionary sorts and formats each term into the return dictionary.

Phases 1 & 2

Phases 1 and 2 process a `.xml` file containing e-mail data into `.txt` files, further processes them into db_load friendly `.txt` files, and outputs the final `.idx` files.

All the processing for phases 1 and 2 are done using Python3 in the main.py file.

The program starts by opening an XML file (specified by the user), and uses and Element Tree to parse through the data. The XML file is processed and four .txt files containing key-value pairs are created, which will further be processed and used in the db_load function.

- terms.txt:
  - The terms.txt file contains every term contained in the emails. A *term* is defined as, "a consecutive sequence of alphanumeric, underscore '_' and dashed '-' characters". The terms are split into two categories, body and subject, having the formats b-"term":row_id and s-"term":row_id respectively
- dates.txt
  - The dates.txt file contains all the dates in the emails, with the format date:row_id
- emails.txt
  - The emails.txt file contains every to, from, cc and bcc field in the emails. The text file has the following format: from-"from_address":row_id, to"to_address":row_id, cc"cc_address":row_id, bcc-"bcc_address":row_id
- recs.txt
  - The recs.txt file contains the full xml data of every e-mail. It has the format row_id:"xml_data"

After the four text files are created, they are sorted using bash scripts, still done in the same python3 file. The sorted files are overwrote onto the original text files.
Next, the sorted text files are reformatted to be used in the db_load function. Every text file is formatted to contain data in the format: key, newline, value. New text files are created for this step.
Finally, the four text files are processed and four index files are created using the Berkely_DB function: db_load.
The db_load command is called with the following options: {-f "filename",-c duplicates=1,-T,-t btree/hash}
Four index files are created: te.idx (b-tree), em.idx (b-tree), da.idx (b-tree), re.idx (hash)

Phase 3

Phase 3 is the data retrieval system. It interacts with the berkeley_db index files that were previously created and outputs data based on the user's queries.

The program has the following functions:

- **next_lex(mystring)** → Returns a string. The search function should return all values that are lexicographically less than this value.
- **terms_search(curs, term)** → Returns a set containing row_ids that have matching terms in the te.idx database
- **terms_search_wild(curs, term)** → Returns a set containing row_ids that have wild-card matches in the te.idx database
- **dates_search(curs, term)** → Returns a set containing row_ids that have matching dates in the da.idx database
- **emails_search(curs, term)** → Returns a set containing row_ids that have matching e-mail terms in the em.idx database

- **body_search(curs, term)** → Returns a set containing row_ids that have matching body and subject terms in the em.idx database
- **body_search_wild(curs, term)** → Returns a set containing row_ids that have wild card matching terms in the body and subject in the em.idx database
- **main()** → Initializes the te.idx, da.idx, em.idx and re.idx databases. The function enters a loop that prompts a user for queries.

C) Testing strategy

The first set of testing we did was ensuring that our .txt files and .idx files had the correct output. Next we tested the “query_creation.py” file could accurately generate the correct strings based on the input from the user. Our implementation involved a dictionary with the key as type (e.g. emails,terms,dates,subj/body), and the value would be the string processed into the correct format (e.g. body:string would become b-string). We ensured that for all inputs, the values would match what would be needed to later access the index files, irregardless of white space or formatting style from the user.
Next we tested our row_id generation for dates, emails, terms, and subj/body query calls respectively. We tested every functionality individually as a single phrase query. For example, for dates we tried using all symbols in the set {‘:’,’>’,’>=’,’<’.’<=’} and making sure that the return values for dates indexes matched the indexes of according values in the dates.idx file. We then tested the intersections of sets between the 4 categories of query calls and made sure the row_id matched the expected row_id to be used for calling the records.
Lastly, we tested using a wide variety of queries including wildcard phrases, various date types, and various numbers of phrases in our inputs. We achieved the expected output throughout our testing.

D) Group Work Strategy

Group communication was done through Messenger. Version control was done using Git+Github.

Name	**Break-down of work**
Bardia Samimi	● Conceptualized and implemented a method to gather row_id’s using set theory ● Search functions for dates, emails, terms ~40 hours
Kamillah Hasham	● Created a robust function to handle and process the strings inputted by the user, differentiating the various operators ● Split the queries so they can be called by the appropriate databases ~40 hours
Neel Kumar	● Prepared files and scripts used by db_load to create index files ● Wildcard functions ● Command line UI ~40 hours
Everyone	● Text file preparation ● Additions to .idx generation script ● Additions to data retrieval script ● Troubleshooting