# CMPT 291 Mini Project 2

*Design Document*

Nebye Berhe (berhe) | Tristan Carlson (tcarlson) | Victor Nguyen (victor3)

## A. Overview of Project

The goal of this project was to work with data in the physical layer. We built an information retrieval system using the Berkeley DB library to query a database of email records. It efficiently evaluates various database queries using algorithms and index files.

This project consists of three phases, each of which contributes to the creation and simple user query of a DBMS. In Phase 1, the read.py takes an XML data file and converts it into four data text files required to make index files. In Phase 2, we take those output text files and load it into an index database file using the index.sh shell script to make hash/B+ indexes. Finally, in Phase 3, the query.py, with the help of query_parser.py, allows a user to make queries. The user inputs (and their conditions) are interpreted as queries for the index files using query.py and display the results.

Our query python script uses the Berkeley DB Python API in order to reference the data within the index files by keys and their values, this is the basis of taking the user input and querying our index files.

---

*User Guide*

---

The XML file must first be put into the data/input folder before running any code. The user is to first run the python script read.py to turn the XML files into four text files titled emails.txt, dates.txt, recs.txt and terms.txt. The user must then run the index.sh bash script to turn the four text files into indexes (hash index for the recs file and b+ tree index for the other three files). The user may then use the python file titled main.py to query various data from the previously created indexes. For example, the user may type in “subj:gas” to find all email records where the word “gas” is in the subject field; to view the full record the user must type in “output=full” following the queries.

## B. Algorithm to Evaluate Queries

Efficiency was kept in mind when creating our phase 2 script for converting text files to indexes by writing in Perl, using minimal lines of code and eliminating as much redundancy as possible. Using hash trees, the main advantage is the use of synchronization as well as increased efficiency versus something like search trees. Using B+ trees for our index files minimizes the number of disk reads (records can be grabbed in an equivalent number of disk accesses), queries for phase 3 were extremely fast due to the data only being stored on the leaf nodes. These heavily impacted our efficacy in querying our inputs for phase 3. Efficiency for part 3 was achieved by only storing data that it needs/will use for what its function pertains

(leaving out anything superfluous) recursively, in theory doing this recursively would be faster/more efficient than running loops and showed true for us during testing.

In our algorithm, a recursive parsing function called rec_parse is used to take a string and return either None or a list of tuples in the form (query_type, val); the function goes through the string to evaluate all the query types (mode, date, etc). If the beginning or entire string evaluates as one of the types, then the function calls itself on the remaining substring, appending the (query_type, val) to the list returned from the recursive call the base case for the function is when string length is 0.

Wild cards within inputs were handled in our main.py and our query_parser.py. When our input in given the main script puts the input into all lower-case letters to match the complete lowercase used in the index files, the input also handles spacing in between the input by replacing it with nothing if found, therefore removing it when handled. Cases where wild card characters (such as "%") were handled in the parser script where if they were found after the end of the characters, they would produce the proper output of the prefix of whatever was before the % sign. Changing the output to full or brief can be done as detailed in the assignment if it is put before the query; this is done through the query parsing script where, for example, if the output is set to full a different return is made in the mode function, telling the other functions to print out all the data instead of the brief forms.

Range searches were accounted for in the algorithm by implementing recursive functions as well as various loops depending on the scenario. For example, in the date query where we had to account for the <,>,<=,>= operators, we looped through a while function until we hit a None type, meaning there was no more leaf nodes containing data to parse through.

## C. Testing Strategy

The testing for this project was done separately for each phase. For Phase 1, Victor had a **diff** command built in with his IDE 'PyCharm'. He used this to compare the output of read.py with the test results given on eClass. The program would highlight the line that is different, which helped Victor pinpoint the error. For Phase 2, to test our shell script **index.sh** result and view our index files. We used the command **db_dump -p** to read and write the results of the index files to a directory. We can then view each index file in a readable text file. Victor also used a shell script checker (reference in README.txt) to double check if the code was in the style shell script wanted.

For Phase 3 we used various index files created from different sized XML/text files (from phase 1/2). Those indexes were queries with various user inputs containing spaces between operators as well as proper formatting to view the output, which consistently came out properly with all the functionalities detailed in the assignment outline. We also tested for wild card related issues pertaining to only the prefix matches. Thorough testing was also made for various scenarios involving multiple condition inputs in which the outputs that we tested did match the conditions set in the input based on the indexes queried.

## D. Group work breakdown

All work was split to what was agreed even and fair between group members. All work was completed to meet the standards of all other group members. Collaboration and aggregation of work was done via GitHub. Communication via text, in person meeting and Discord chat.

The Breakdown of who did exactly what work is listed below but note that overlap did occur between group members for error checking, aggregation of independent works into one finalized unit and bug fixing.

- Phase 1: Victor Nguyen
- Phase 2: Nebye Berhe / Victor Nguyen
- Phase 3: Tristan Carlson / Nebye Berhe / Victor Nguyen
- Project Document: Victor Nguyen/ Nebye Berhe
- Testing: Tristan Carlson / Nebye Berhe / Victor Nguyen