General Overview of System (How to run this project)

There are three folders in the project submission. Phase1, phase2, phase3. The appropriate files for each file must exist in each folder. A given phase will only write files inside the current working directory.

To run phase 1, cd into phase1 dir, NEXT, run the command ‘make’ this will make the executable ‘parse’. The contents of the xml file must be piped into the parse executable as such:

Cat example.xml | ./parse

The outputted .txt files will be created in the directory after about 4.5 minutes if there are 1million lines.

To run phase 2, move the .txt files from phase1 folder into the phase2 folder. Cd into phase2. Run the command

./phase2.sh

Which does 4 parallel pipes, sorts, perl scrubbing, db_loads to create the idx files in the phase2 directory. This should take at most 30 seconds.

To run phase3, move the idx files from phase2 directory into the phase3 directory and then run the command

Python3 phase3.py

Which will start the CLI to the database. To exit the database type in the command exit() or hit Ctrl+C/

Detailed Design of Software

Phase 1: C++ file. The parse file takes in stdin line by line and processes each line in roughly O(n) runtime where n is the number of chars in the string. Once the input and output filestreams are initialized the first two lines are stripped off of the file and the program moves into the main loop, taking in each email line in the file. First the row identifiers are found and the row number is taken out from in between them. Likewise, the subj, to, from, and all other fields are pulled from the string. Now that we have the fields we can start to parse them into their appropriate txt files using the row number as an index. The easiest are the date and rec files, passing in the

row number along with the date field or the whole text string respectively. The emails have to be separated using commas if they're not empty and put in the email file in the correct order. The subject and the body are then put into th term file, parsed by stepping through each char in the fields and logically separating out the valid terms. The next line is then called for the next loop. After the file is looped through we're done, close the files.

Phase 2: a Bash script that sorts each .xml file, creates the indexes and the proper data structure for each Berkley DB and then outputs each index into a seperate .idx file

Phase 3: Several different python scripts. dates.py takes in a list of queries, where the first one is guaranteed to be a date query. It then verifies if the date query is valid. From there it will output the rows of the corresponding .idx file that satisfy the demands of the date query. The approach for emails.py and terms.py is effectively the same. phase3.py determines what time of query is at the start of the query string, it is also responsible for determining the mode and printing our outputs of the queries. bdb_helper.py provides a basic interface and helper functions for loading the database.

Testing Strategy

For each parsing section, we developed random inputs to meet specific case requirements and would print the outputs to ensure we were getting the correct output for each corresponding case

A description of your algorithm for efficiently evaluating queries

Queries are separated into three lists, one for date queries, one for term queries, and one for email queries. Each list is then passed on to a processor function which handles all of the same type of query. For example, all the term queries are passed off to a single function. Each pass through the processing loop generates a set of row numbers that satisfy a single term query. We then intersection update the global set of rows with the current set of valid rows. So, in the end, n set joins have been performed (where n in the number of term queries) ensuring that the returned row numbers satisfy all of the term queries.

Email queries are done in a similar fashion to term queries. For n email queries, return n sets which each represent the valid row numbers. Then set intersect all n set. This operation is close to linear / constant runtime in python.

Both emails and queries are log(n) + i database accesses to find the first possible matching element, the i more accesses to iterate until the cursor does not match the string in question. That operation is applied x times for x term queries then constant runtime set joined in the end.

Date queries are simple, there is a mathematical algorithm which combines an arbitrary number of date ranges into a single date range. We combine that condensed date range with the range of the database, which returns a range of dates that exist within the database, within all the date queries. We then iterate linearly from start to end of that range. Hence, O(2 log(n)) + O(i-j). Two log operations to find the start and end data points, then iterate over the dates in between the start and end point using cursor.next() which is technically a log(n) access with b+ trees.

Once each query has been applied to the corresponding date, term, or email b+ tree, each type of query has returned a set of valid rows, those sets are then set intersected. For example, a set of all rows that satisfy the combined term queries is intersected with a set of row numbers which represent all the rows that satisfy the combined date queries. This then produces a final set of rows which is then passed on to the records hash database file which linearly prints off the full or condensed record corresponding to each row.

Group work break-down & coordination

Task break-down

Phase 1: Most of the code was programmed by Josh. Josh and Alex pair programmed at the start of phase 1 and Josh completed it on his own

Phase 2: Most of the code was programmed by Alex. Chris and Alex pair programmed at the start of phase 2 and Alex completed it on his own

Phase 3: Alex and Chris discussed how to approach the entirety of phase 3 beforehand and updated Josh later. The parsing was spread equally throughout all group members. Josh and Alex did most of the debugging at the end. Alex and Chris did the final testing

Group work strategy and/or plan

All three group members met up and programmed everything all at once together at the same time using git