**CMPUT 291**

**Mini Project 2**

**Report**

Dylan Alcock: Lab H01

Adam Otto: Lab H05

Austin Crapo: Lab H05

Collaboration:

Did not collaborate with anyone else.

General Overview:

For phase 1 we ask the user for an XML file with tweets in it and go through the text file extracting the tweets, terms, and dates from that file. Then removing all special characters from everything that was extracted. Once all of the characters have been removed tweets.txt, dates.txt, and terms.txt by adding the correct key and value to the files, splitting them with a ‘:’.

For phase 2 we take the three text files and use the UNIX sort command to sort the files and only keep the unique entries. With the sorted files we run the break, Perl script given to us and split the text files so the key value pairs are now on separate lines as db\_load expects. Then by running the db\_load command on the new files, with flags to take the text file and accept duplicates we create the index files used in Phase 3.

For phase 3 first we open our index files and create cursors for all of them. We start with a welcome message and then get the user input of the query they would like to enter. We then run the query(s) and output the tweet IDs, along with all the other information of the tweet displayed in a human readable format. Parsing through the queries we get the index that the query needs to be run on, retrieve the data from the indexes. If there are multiple queries we then intersect the IDs and then find the IDs and print out the tweet data.

Algorithm:

For evaluating queries it generates a machine readable description of the query, then parses the list of queries and removes redundant queries, finally it will sort the queries based on the likelyhood of that query returning large result sets (so that small result sets are evaulated first). Then it will fetch a result set of IDs for each query individually and intersect the IDs together query by query. Since the queries are sorted and the Python set intersection time complexity is on average O(min(n,m)), we genereally keep our time complexity to O(n) but in the worst case it could be O(n^2). Then we print the IDs that intersect from all the queries and display the tweets. For wildcards (in terms) and ranges (in dates) the algorithm finds (using the Btree) the first element that matches the search and then uses the sorted aspect of the database to simply run through until finding the last element of the range. The efficiency of our algorithm is above average as it avoids searching the entire database at all costs, and keeps it's running result sets a small as possible to avoid memory overflows.

Testing Strategy:

For out testing strategy while making the function we had used our own methods for the parts we had worked on. Mostly using print statements within functions to see how our functions are working and what the potential output would be. For final testing we would run queries of all the types as specified in the spec and then test extreme cases where lots of queries would be entered or where lots of output would be printed.

Group Work Strategy:

For our work strategy initially it started as one of use doing phase 1, and then phase 2, then for phase we had specific parts we would work on such as parsing the input queries or retrieving data from the indexes. We used GitHub as our main way of working and sharing code. We had talked in class and over text message what we had worked on and parts of the code that were working or not working so well.

Any Assumptions: We assumed that for testing purposes we would need to print the number of results returned, and have a function to gracefully exit the program. We also assumed that at least two querie result sets worth of ID's could be held in main memory at any given time, even if those result sets were both all ID's in the database. We also assume that a search for a keyword (ended properly with a ':') that has an empty keyword (e.g. 'text:') should return nothing.

Dylan: Created the index files for phase 2 (1 Hour), and worked on data\_retrieval in phase 3, initially creating all the information retrieval functions, and making most of them run with all test cases (15 hrs)

Adam: Created the first draft of the phase 1 data file preperation. Updated phase 1 to read a file line by line instead of using the tree command. Created the display function that would print out the tweet information in full from the tw.idx file. General debugging and testing on data retreival functions.(15 Hours)

Austin: Created the Phase 3 parsing code that broke down queries (5 hr), created the spec for creating conditions from queries, then wrote the code that would clean and parse a conditions list (5 hrs). Various bug testing and cleaning of code, reworked data retrival functions to better use Berkely functions (4hrs).