Overview

This goal of this project is to keep data in files and maintain indices that provide basic searches over data. In this project, we use the Berkeley DB library to operate on file and indices. Here is the user guide: The user first has to run phase1.py with the file name following that argument to read records in xml from the standard input and produces 4 files with the following name: terms.txt, pdates.txt, prices.txt, ads.txt. Then the user has to run phase2.py to sort the files and keep the sorted data under the same file names. Lastly, the user runs phase3.py, then the user enters the program, the user can input some searching conditions based on price, date, location and category (reserved keywords) and any keywords that might be related to the expected result (The data are ads posted from kijiji). The date can be exact date, (i.e date = 2018/11/07) or it can be range search as well (i.e date>2018/11/09), the date format has to follow yyyy/mm/dd. The same concept applies to price, it can be exact price(i.e price = 40) or it can be range seach (i.e price > 40). The user can enter multiple searching conditions at once, (i.e price > 40 location = Edmonton cat=bed-mattress date>=2018/11/03 date<=2018/11/07) We would print out all matching results with the ad id and the title of all matching ads. The user should be able to change the output format to full record typing “output =full” which means every information related to that ad would be printed out and can also go back to only printing out ad it and the title by entering “output=brief”. The user always has the option to exit from the program by typing “exit”.

Algorithm Description

Queries with multiple conditions:

When evaluating queries with multiple conditions, the main idea is to compare the number of arguments to the count of matching id for each argument. For example, if the user input is as following:(camera date>2018/11/03 date<=2018/11/06, location= Edmonton), as you can see, there are four arguments in this user input. We only want the result that matches with all four of those conditions, for the first condition, camera, we go to the terms file and search through all the terms then get the id of all matching results, and store ids in a list and the count for that id + 1. Then we go on evaluating the next argument, date> 2018/11/03, we go to the pdates file and grab the ids of all matching results and store it in the same list. And the count +1 for each matching id. Then we go on evaluating the next argument, date<=2018/11/06, we go to the pdate file and grab all ids of all matching result and store it in the same list. And the count for each id +1. Next, we evaluate the last argument location = Edmonton, since we already got the matching id from the pdate flie, we can go to that file and grab the matching result for location=Edmonton and store all matching id in the list again and do count+1 for each ids. At the end, we compare the count of each id in the list to the number of arguments of the user input. In the example above, we only print out the id that has the count of 4.

Queries with wild card:

The user may input something like camera%, its different from when we input camera. For camera, the result has to be the exact same word. For camera%, this is what we called partial match. In order to solve this problem, we have a parameter called isExactSearch which can be true when the keyword does not contain % or false when the keyword contains %. If isExactsearch is false,(i.e camera%), we use.strip( ) to get rid of the % and use the startwith( ) function in python to iterate through the whole terms.txt file to see if there is any matching result. If isExactsearch is true (i.e camera), we simply user.set( ) to find the matching result and iterate through the whole terms.txt file.

Queries with wild range_search:

The main idea for range_search can be divided into two parts. One is for "< " and "<=", in those two scenario, we compare the user_input date to the first date in the pdate.txt file which is the smallest date. If the user_input date is smaller than the smallest date in the sorted pdate.txt file, that means there is no matching result and we would just return an empty list. If not, we iterate through the whole pdates.txt file to get all matching result and store the matching id in the output list. (the range_search for date and price uses the exact same logic, so in here I am only explaining using date). Another case is for ">" and ">=", we use .set( ) to see if there is any matching date in the pdate.txt file. If there is, we then use .next( ) to get all dates following that matching date and store all of them in the output file. If there is not, we use strptime( ) to split the user_input date into three parts, yyyy/mm/dd and we add 1 to days part until it matches one date in the pdate.txt file that is greater than the user_input date. Then we use .next( ) to get all dates after the date that is matched above.

Analysis of effiency:

The location and category search can be treated as weak entites, since it has never been used as id in either of the four files we created above. The main idea to have better running time on this project is we treat location and category when it is searched solely or searched with other conditions that has been used as a key. For example, we treat the following differently when it comes to evaluating the queries, (user_input: cat = bed_mattres// user_input : cat= bed_mattres date >= 2018/11/06), for the 1st example with just cat=bed_mattres, this one we has to go for the function QueryByfunctionSlow( ), which means we have to manually go to the price.txt file to iterate through all the location information with index 2. For the 2nd example, we would use the range_search on date to get all matching ids first, and for the cat part, instead doing the slow method above, we would use dictionary here to just iterate through the result we found for the date argument. This would increase the efficiency of our program dramatically.

Testing strategy

For the testing strategy, we use 10.xml when we trying to see if the logic of our range_search is right or not. When use 20k.xml for the final check. We would pick one specific id and input as many conditions as possible to see if it gives us that one exact correct answer.

Group work breakdown strategy

Chongzhe: range_search for price and date and report (time spend 7.5 hours)

Zijun: phase 1 and strip& convert function and location&category search function (9 hours)

Ruiqin: phase 2 and queryevaluation function and initializedict function (9 hours)

We gather in the lab room and discuss what we had and what we still trying to figure out every day after class.