Overview

User Guide

To begin, our system is initialized by running *phase1.py* with the records file as an argument. After, use the Linux *sort* command on each of the output files generated in phase1. Then, *phase2.py* must be called to create the Berkeley DB files, without any arguments. This completes the initialization stage of the system and once *phase3.py* is ran, the query terminal is opened as the user may query the databases.

With the terminal open, the user has the option to enter a query or access the help screen by entering *!help.* Queries consist of any number of the following types of subqueries

Price Query	price [<|>|<=|>=|=] ###
Date Query	date [<|>|<=|>=|=] ####/##/##
Term Query	*any word with - or _*
Category Query	cat = *category*
Location Query	location = *city*

System Overview

The first two phases of the system use an iterative approach to extracting information from the records file and creating the database, operating on a line by line basis. Considering the third phase, we created a query parser and a database manager. The query parser is responsible for extracting the different subqueries from the main query and returns the subqueries to our database manager which evaluates the query and efficiently searches through the database files for indexes that satisfy the user's query. The results are then displayed in a table in the terminal and the user then has the option to enter another query.

Algorithms and Evaluation

After retrieving the user's query, our system counts the number of each type of query entered and stores each type of subquery in a dictionary of lists. Based on the length of each subquery list, the manager evaluates which index file to query. For example, if the query consists of a price query, category, and location, the price index file is queried and at each index the location and category restrictions are considered and added to the results – a list of sets. If the user enters the above subqueries but also specifies a term query, the database will also traverse the terms index and select and match and ads that contain all terms. These results are added in another set to the results list. After all subqueries are evaluated the results list of sets is intersected, and those ad IDs are retrieved from the ads hash table and the output is printed from the table. If the user enters only a location or category query, the database will traverse the entire dates index and match all ad IDs that satisfy the requirements.

A term query with a wildcard uses regex to match any terms that partially match the query and any succeeding characters. A range search is made to be efficient by having a separate function to query based on the range. Each index is read from the database starting from the lower bound and evaluated based on both restrictions on the date or price and will return the results as soon as it exceeds the upper bound. This way, the indices are only traversed once; effectively halving the query time.

Testing Strategy

Our testing strategy for phase 1 was simply looking at the text file that was created and comparing and contrasting to the reference tables that were provided. We used the windows file compare command, FC to find the differences between our tables and the reference tables. After finding differences we were able to go to the original file that we were sorting from and look up the location where our results were wrong. This often made it apparent what the issue was, because we were able to see the string containing something that we had not expected and made a case for. Testing phase 2 was done with using some of the "iteration" code from the slides to go through the databases and print out what they contained. This way we were able to put the code into each function and see their results.

Our testing strategy in phase 3 evolved from testing trivial base cases to testing complex edge cases. While creating the database we tested it in batches, such that whenever a method was "complete" we tested the trivial cases. The trivial cases we used were for testing the base functionality of our program. Using the queries from simple cases that were easy to follow to ensure that they were correct. Utilizing the smaller test table to be able to know each entry, so that it was clear what each query should be returning. These queries did not often present any issues but proved that our code and logic were functional. Eventually as more of our program was developed, we began to use more complex test cases. These test cases started to involve thinking of edge cases. Testing the boundaries of the queries, the literal edges of the files. Testing complex queries that involved multiple queries. We used a counter to show the amount of results in the table, so that it was easier for us to see if our results matched what we knew was in the table. Once we knew that it was working on the smallest database that we were familiar with we started testing it on the larger datasets that were provided. These took more time to cross check that our results were correct.

Group Work

Our group work strategy was to divide and conquer the different aspects of each phase of the project. For example, in phases one and two, both partners took on the parsing and writing of half of the output files. Considering the third phase of the project, one partner created skeleton code for the database management portion of the project, and both partners tackled the individual methods within the code.

Jesse Goertzen – *jgoertze*

- Phase 1 and Phase 2 – Prices, Dates, Ads (~4 hrs)
- Phase 3
    - Parser – (~2 hrs)
    - Portions of Database – (~11 hrs)
    - Terminal (30 m)

Zach Kist – *zkist*

- Phase 1 and 2 – Terms (~6 hrs)
- Phase 3
    - Portions of Database – (~10 hrs)