Name: Mark Tupala
Student ID: 1188594
Unix ID: tupala
Course: CMPUT 291
Assignment: 4

In assignment 4, we were tasked with using a Berkeley DB implementation to store information containing users' ratings of songs, and then query this database for provided songs to determine the three most similar songs in terms of users' ratings.

Implementation Details

I decided to use Python, because I am familiar with it and believe that it is a highly productive language. The Python standard library contains a module named 'shelve', which uses an underlying database to associate string keys with Python objects as stored values. The only difference between using shelve and the underlying database itself is that the shelve module utilizes a serialization algorithm to convert Python objects to and from strings that can be stored in the database. While the shelve module is database independent, it can be determined which underlying database is being used by using the 'whichdb' module, also included in the standard library. The whichdb function accepts the filename as an argument and returns which type of database the file is. On the lab machine I tested my code on, calling this function with the databases I created returned 'dbhash'. According to the documentation for the dbhash module, “The bsddb module is required to use dbhash”, and according to the documentation for the bsddb module, “The bsddb module provides an interface to the Berkeley DB library...The bsddb module requires a Berkeley DB library.” Therefore, the databases I created are Berkeley DBs.

Performance

One of the goals of the assignment was to use an index to speed up the queries. When doing a linear search through the main database, every song must be checked to determine if one of its raters has also rated the queried song. To speed up queries, I created a secondary database that stores the raters as keys, and all of their ratings for songs as the stored value. Therefore, when given a query for a particular song, the program only has to look up the raters of that song from the main database, and then look up which other songs those raters have rated in the secondary database. In other words, by using this index, every single song that is checked will have a computable distance from the queried song. No song will be checked that does not share a rater with the song being queried. This led to a dramatic increase in speed. Below are the results I obtained on a lab machine using two data sets of different size, which I have included in my submission:

10000 songs, 1000 users, 10 ratings per song, 10 queries:

linear: 1.57 secs (0.157 secs/query)
indexed: 0.03 secs (0.003 secs/query)

100000 songs, 1000 users, 10 ratings per song, 10 queries:
linear: 16.08 secs (1.608 secs/query)
indexed: 0.24 secs (0.024 secs/query)

Design and Testing

Included in my submission are five Python source files. 'create_data.py' is a very simple module that I used to create random data sets, including the submitted ones. It accepts three command line arguments: the number of songs, the number of users, and the number of raters per song. 'create_db' is used to parse the resulting text file and create the main database and the index with the data. 'common.py' includes functions that are used by both 'linear_search.py' and 'indexed_search.py' and includes a function for randomly generating 'queries.txt' files.