University of Toronto
csc228 — File Structures and Data Management, Fall 2001

Assignment 2 — Direct Access for Query Processing

Part A Due: at the beginning of class, Monday, October 15, 6:10 pm.
Part B Due: at the beginning of class, Monday, November 5, 6:10 pm.
Total weight: 15% of your course grade. Part A is worth 10 % of the total weight.

You will do the assignment in two parts. Part A is merely a step on the way to part B. It will be handed in for marks simply to encourage you to get an early start on the design.

The purpose of this assignment is solidify the concepts of:

1. constructing a sorted index from a large record file on disk.
2. searching a sorted index file.
3. cosequential processing to match (intersect) two files.
4. basic interactive i/o

Problem Statement

Good news! Napster appears to have reached a licensing deal with the US music publishers (see http://www.torontostar.ca/ and search on Napster). That means your venture is even more viable. However, you realize that your product still has a ways to go. Although you've provided (in Assignment 1) a means for bundling up songs by album, you haven't provided a query mechanism. Furthermore, you've also decided that, unlike Napster, you'd like to be able to search for songs or albums by genre – one more attractive feature that should bring Napster users over to your product.

Your task is to build a query mechanism that supports the following queries:

1. Get all songs by a specific artist.
2. Get all songs in a specific genre.
3. Get all songs by a specific artist in a specific genre.

To support the first two queries, you will need to generate one index that maps artists to songs, and a second index that maps genres to songs. Although there is enough room in memory to build an individual index, there is not room to hold them both in memory. Therefore, once you build an index, you then need to write it out to an intermediate file. At the end of the index construction process, you will have two intermediate files, each containing an index.

For the first two queries, you will need to search the appropriate index file to find out the entry for the query artist or genre. That entry will point to a collection of song file relative record numbers. For the artist query, the output will be a set of (song title, album) pairs, while for the genre query, the output will be a set of (song title, artist, album) triples. If the query artist or genre has no songs in the song file, nothing will be output. The third query is a bit more complicated and will require that you find the intersection of an artist query and a genre query. To do this, you will create a temporary file containing the relative record numbers of the songs by the query artist and a second temporary file containing the relative record numbers of the songs of the specified genre. Finally, using a cosequential match process, you must compute the intersection of these temporary files. The output will be a set of (song title, album) pairs.

Assignment details

Program Interface

The song input file is the same input file you used in Assignment 1, and will be the only parameter to your program. You must ensure that your input file has a fixed record length, since you will be using relative record numbers (RRN's) in your assignment. Once your program has built the two indexes from the song file, it will accept queries from standard input and write the results to standard output. A query can be of the following form:

1. an artist query of the form <artist>. For example, at the prompt, the user would enter "Neil Young" (without the quotes) and the program would then display the title and album (but not the other information in the song record) of all Neil Young songs, one per line. You can assume that the domain of artist names does not intersect the domain of genre names.
2. a genre query of the form <genre>. For example, at the prompt, the user would enter "rock" (again, without quotes) and the program would display the title, artist, and album (but not the information in the song record) of all rock songs, one per line.
3. a compound query of the form <artist>&<genre>. For example, at the prompt, the user would enter "Neil Young&rock" (without quotes) and the program would display the title and album (but not the information in the song record) of all Neil Young rock songs, one per line.

The output format will be the same for each of the above query types, i.e., one song title per line. Note that there should be no spaces before and after the "&" in the compound query. The original input song file must be left intact by your program, and your program must remove any temporary files that it creates. You need not do a lot of error checking in your program, but you should specify in your report what error checks would ideally be done, and which function would be responsible for each.

Overall algorithm

Your overall algorithm should:

- Read in the song file, record by record. For each song record, extract the "artist" and "genre" fields. Use these fields to construct one index sorted by artist and a second index sorted by genre. You will not be graded on the efficiency of your sort algorithm. Using appropriate file structures, each index is then written out to a temporary file (or files). **NOTE:** I want your index file structure to be space efficient, i.e., minimize internal fragmentation.
- Given an artist or genre query, you must perform a binary search on the corresponding index file. If the query item does not exists in the index, you should simply output nothing.

- Given a compound query, you must compute, again using binary search, the solutions to each of the separate queries, and store them in temporary files. Next, using a suitable cosequential process, compute the intersection of these two files and generate the query response. All intermediate files must be removed. Again, no solution means no output.

You have much more freedom in this assignment in the way of data and file structure design. Make sure you document all your decision decisions.

Developing your program

As was the case in Assignment 1, I recommend that you implement the program in phases, testing your program at each phase before moving on. You should be able to use much of your i/o code from Assignment 1, thereby making the i/o task much easier. Remember that the song file must remain intact.

What to hand in for part A

There is no specific requirement for Part A; it is merely a step on the way to part B. I'd like to see your class hierarchy, showing member variables and member functions for each class. I'd also like to see a pseudo-algorithm for your program, and how you plan on decomposing the task into simpler subtasks. This is probably the most important step in a design (the coding really is the final, rather mundane step). You should have made a good start on the program, and have some components working. You may need to create simple driver programs to test your code at each phase of development. Hand in a printout of your code (including any driver programs), and some test runs that demonstrate what works. Your code needn't work flawlessly for you to get some marks on part A. Do not hand in a report with part A.

Your tutor will evaluate your work on this part by looking at your object design, your algorithm, and your test runs (however simple), so be sure that they demonstrate what you have working. Explain your test results with brief, hand-written comments on your printout.

What to hand in for part B

For part B, you are to hand in your assignment in the form of a report, as described in the 228 Course Guide. Your report must include all of your code and all relevant input and output test files. Remember to justify the decisions you made in designing your program. This is a very important part of your report. Your report should be thorough, well-written, concise, and honest. It should be no more than about five pages long, not including code and tests. Make sure that you run your report through a spelling checker.

You are to devise your own testing for this assignment. Although the program is designed to handle a HUGE input file, you can test it on relatively small ones. You will be marked for testing according to the standard set by the document on "Software Testing", available on the course web site. Also note that if your testing is weak, you will lose marks not only for testing but also for correctness; the only way we will know that you program works is if you *demonstrate* that through your testing.