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

Project — Napster Finale

Project plan due: *Monday, November 12, 6:10 pm, in class*
Presentation: *Monday, November 26 and Monday, December 3, in tutorial*
Final project due: *Monday, December 3, 6:10 pm, in class*

Introduction

In Assignment 2, you designed a system that allowed you to query song records by artist, by genre, or by both. You had the luxury of mapping secondary keys (e.g., Neil Young or rock) directly to song record RRN's, and you didn't have to worry about adding or deleting song or index records. You were also content with $O(\log n)$ secondary key index search complexity using binary search (on an array). For this project, you will extend your system to include a primary key index, to accommodate larger files, to accommodate more efficient secondary key indexing (through hashing), and to accommodate efficient primary key indexing through B-trees.

Rough specifications

Key Components

The main components of the project include:

- Once again, you will be working with an input file of unsorted, fixed-length song records.
- You will support on-line addition and deletion of song records to/from the song record file. You will maintain an available record stack (implemented by a linked list) in the song file. That means that your input song file should have an available list head pointer as its first record. Let the head pointer record be a fixed-length record containing a single, fixed-length text field of size 10 characters containing the relative record number of the first available record (or -1 if no such record exists).
- For each song record, you must define a suitable primary key and hold all primary keys in a sorted primary key index. Your primary keys must be chosen in such a way such that when the song records are sorted by primary key, they are sorted by artist name followed by (in the case of same artist name) album name followed by (in the case of same artist name and same album name) song name. You may assume that the string formed by the first three characters of the artist name followed by the first three characters of the album name followed by the first three characters of the song name (including, in each case, blanks) is unique.
- Since the primary keys cannot fit in memory, you will organize the primary keys in a B^+-tree and write the B^+-tree to a primary key index file (which should be deleted upon program termination). A B^+-tree is simply a B-tree with a linked list joining the leaves together to

facilitate sequential processing of the leaves. The B^+-tree order will be an input parameter to your program. When a song record is added or deleted, you must perform the corresponding primary key addition or deletion from the B^+-tree.

- You will again support queries of the form: "Show me the songs recorded by Neil Young" or "Show me all the rock songs" but **not** compound queries of the form "Show me all the rock songs recorded by Neil Young." From the input song file, you will construct a hash table in memory that maps an artist name to a location in the table. At that location will be a collection of secondary keys that hash to that location (recall that there may be collisions, so that at a given location you may have multiple artists). Each secondary key, in turn, will point to a collection of primary keys. Note that although there is room in memory for the hash table and secondary keys, there is not room for the collections of primary keys, which must be stored in a file (which should be deleted upon program termination). You will construct a similar hash table for genre queries. You have lots of room for design flexibility here. You will need to choose a method for storing (in memory) your collection of secondary keys, and a way of storing (in a file) a collection of primary keys for a given secondary key. Justify your design with respect to the task at hand. For each of your hash tables, you will decide on a hash function and collision resolution scheme. When a song record is added or deleted, you must perform the corresponding primary key addition or deletion from your two hash tables. The hash table bucket size will be an input parameter to your program.

- You will support output of the song records sorted by primary key.

Operations

You must provide at least the following operations:

- list all song records, sorted by primary key. This would be a query of the form (at the prompt and without the quotes): "list". This is a good diagnostic to make sure that your B^+-tree is sorting the primary keys properly. Note that for any set of song records, there is a unique sorted list (see sorting convention described earlier).

- add or delete a song record. This would be a query of the form (at the prompt and without the quotes and angle brackets): "add:<song record>" or "del:<primary key>"

- 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. As before, you can assume that the domain of artist names does not intersect the domain of genre names.

- 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.

- you do not need to support the compound query from Assignment 2.

Note that adding or deleting records from the song file will permanently change the file. If you want to be able to test your program with the same song file again, make sure you make a backup copy of it, so that you can restore it from the backup after it has been changed. Note that your program should 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. Make your hashing and B^+-tree code general, so that they can be easily adapted to another domain, a different collision resolution scheme, etc. Remember that B-tree order and bucket size are input parameters to your program.

How much you must accomplish

Groups of two must implement at least the specification described above. Groups of three must also accomplish one or more of the following tasks:

- In addition to your B^+-tree, implement a variable-order simple prefix B^+-tree. Compare this to your original B-tree in terms of space requirements.
- Implement an incremental hashing scheme, i.e., a scheme for restructuring the hash table when the load factor has become too high and performance therefore has dropped. See chapter 12 of the text. Large order means shallower trees and more efficient search. However, larger order also implies greater depth in the local binary search performed at a given node to find a key. Run experiments that vary node order and plot the total number of operations as a function of B-tree node order. What conclusions can you draw between B-tree order and search complexity?
- Performing experiments to compare the performance of various hashing schemes, i.e., comparing various kinds of hash functions and/or collision resolution schemes.

I won't tell you what or how much extra work to choose. Aim to make your total workload per group member roughly what it would be in a group of two that is doing only the basics listed earlier.

Program interface

Your program will take the following input arguments in the following order:

1. song file (remember that this now has an available record top of stack pointer as its first record). Note that the maximum number of songs in the song file is 1000.
2. B-tree order. Note that the maximum B-tree order is 10.
3. hash table bucket size. The maximum bucket size is 10, the maximum number of artists is 100, and the number of genres is fixed (same as in Assignments 1 and 2).

Once invoked, your program will take commands from standard input with program execution terminated by cntrl d. Note that all files that you create should be deleted upon program termination.

Working in a group

You will complete this project in a group of two or three people. Groups of three will be expected to complete a more ambitious project than groups of two, as described above.

Be sure to reread sections 5 and 6.2–6.3 of the 228 Course Guide; they address how to set up and organize your group, and how your group work will be marked.

I hope that working in a group will be one of the most valuable experiences you have in this course. However, you will have to make an extra effort to deal with the dynamics of a group if you wish it to be a valuable *positive* experience. `:-)`

If your group is having difficulties working effectively, don't sweep it under the rug and hope things will improve; they probably won't. Talk about it, and if that doesn't work, see me as soon as possible.

Using the code from your textbook

The appendices of your textbook contain C++ classes that partially implement B-trees. You are welcome to use this code by adapting it as necessary for your project. Because many of the B-tree functions are already implemented, using this code may save you some time. However, there are some hurdles you will have to overcome in order to use the code:

- The code uses objects and methods from other classes presented in earlier appendices. These include SimpleIndex, IOBuffer, Record and possibly others. In order to understand the B-tree code, you will need to understand these other classes.
- It does not implement deletion. You would have to add this, and it is the hardest part of the B-tree code.
- It does not quite implement a B^+-tree, in that it does not link the leaves together. You would have to modify the code to add and maintain these pointers.
- **There are bugs in the code.**

In the real world, programmers often have to make decisions about when to use and adapt "legacy" code and when to create it from scratch. You need to consider this decision as a group as you plan your project.

The phases of the project

Your project plan

Your project plan should include the following information:

- **Group**
  The name of your group, your group members, and your tutor. Write this on the project cover sheet, available on the course web site.
- **Overall Design**
  Describe your overall design in terms of its constituent processes. Describe the sequence of operations you expect to perform for each of the commands outlined above.
- **Data and File Structures**
  For each set of data, a description of the data or file structure used. Will it be hashed, or indexed?, what sort of collision resolution scheme will be used?, will it be linked to any other file?, etc.
- **Plan of Attack and Schedule**
  A detailed breakdown of tasks. For each task, specify who will focus on it, its planned completion date, and whether it is core or a fancy extra that you hope to have time for.

  You can think of the work to be done as a two-dimensional table, with components of the program down the rows and phases of development (settling the specifications, design, coding, testing, report writing) across the columns. It is acceptable for group members to each focus on different aspects; however, it is **not acceptable** to divide things up simply by row or column – for example to have one team member do everything to do with hashing, or everything to do with writing the report.

The plan of attack and schedule is probably the most important part of your Project Plan. Most teams neglect it and later suffer for that.

You will be allowed to change your mind later about anything in your project plan, but of course, the better thought out your plan is, the fewer changes will be necessary. Be prepared also that your TA may find flaws in your plan, and that this may lead you to make changes part way through the project. Accommodating changes will be much easier if you have designed your code well.

I would rather you get a modest project done really nicely (including robust code, thorough testing, and well-written report), than over-extend yourself and end up with a very ambitious project done poorly. And yes, a well done but modest project will get a better mark.

A wise strategy would be to plan to complete a modest project, and then if you can, to improve or extend it. Explain this strategy in your Project Plan (including exactly how you hope to extend your project). In order to work this way, you *must* design your code well, so that pieces can be plugged in and out.

Your group presentation

During the last two tutorials, each group will make a brief (roughly 5 minute) presentation of their project, discussing their overall design strategy and special features of their system. In most cases, the group should select one member to do the actual speaking, but every member should participate in planning the presentation.

We will say more about the presentations in an upcoming tutorial.

Your final report

Your final report will be significantly longer than the report for assignment 2B, but shouldn't be longer than about 10 pages.

Include the usual sections of a report, as well as all the sections listed under the project plan above. It is okay to reuse parts of what you handed in for your plan, but you should update it to reflect changes to your plan, and you should expand upon the design decisions you made, alternatives you rejected, tradeoffs, etc.

In addition, you must submit the following sections:

- **Who did what.**
  Each group member must submit their own brief assessment of who did which work on the project. Even if your group writes this "who did what" statement together, each person must hand in their own copy. These statements should be brief — substantially less than a page long.

- **Teamwork.**
  A brief description of how you handled the teamwork, beyond who did what. Was one person in charge of everything? Was a different person in charge of each of several sub-tasks? Or was it a more democratic team structure? How did you run your team meetings? What problems arose in your team? How did you solve them?

Approaching the project

You will save yourself a great deal of time and effort overall if you take the time to do a good job in the early stages of the project. You should have a clear specification of *what* you plan to do before beginning to do it. And you should design your code carefully from the beginning, rather than think that you will reverse-engineer a good design after it's done. Don't forget the lessons learned in CSC148 and beyond about modularity, abstraction, and information hiding.

You may find it necessary to reduce your ambitions, and you may learn things later that lead you to change aspects of your system. For both reasons, it is important to begin by isolating the essential operations of the system and designing a modular structure to accommodate later changes. This is not a difficult task, but you must remember not to rush straight into coding. If, on the other hand, you find that you have quickly succeeded in building a basic working system, you may wish to add extra features to your program.

It will be natural as you plan your system to think about how to describe it in your report. The report should **not** be written at the end of the project. Rather, you should write an outline at the beginning, and continue to expand it as you proceed, until by the end of coding and testing, the report is almost complete.

You should also begin work on your testing strategy very early in the project. Remember, many of you lost marks for incomplete testing in Assignment 1. If you leave testing to the last minute, your group will likely find itself in trouble. Try to build up the system from a simple skeleton to a fully functional package in gradual stages, testing various components as you proceed. With the confidence of having something simple working soon, you can go on to grow a bigger project successfully.