Homework 1

Dillon VanBuskirk
dtvanbus@uark.edu

Jaewung Ryu
jryu@uark.edu

September 10, 2016

Contents

1 Introduction

We coded this database system assignment using C++ with the goal of practicing the file management techniques. In this assignment, it provies a menu for end users to perform a multitude of tasks on databases. It allows users to create a new database, open and close a database, add or delete records within a database, update and display a record, as well as create a human readable report that is written to a seperate file of of the first 10 records in a database.

This program does not use any memory for storing the records and uses files for all of the storage and operations. It reads and writes fixed length records from and to a file, respectively. It uses a multitude of techniques to get the required information from the file in order to properly behave like a database. One thing we did not support was after deleting a record, we did not allow users to create a record in the place of it. This was because the method of deletion that we decided to go with, which was replacing the values of each field with a null or otherwise error value and keep the ID the same.

2 Overflow Handling

We are asked to discuss our method of handling the overflow - that is, each new addition of a record. We decided to handle overflow by sending all newly created records to a secondary file titled overflow. This is handled by the program so that when a database is opened, it creates a blank file called overflow and each

newly added record gets written into it. When the overflow becomes too large, it will be merged and subsequently sorted with the database. This was decided by the assignment to be a maximum of 4 records stored in the overflow at a time. That is, when the fifth new record is added, it calls the linux command line to perform a merge and sort. This way, linux can perform the sort on the files for the program as opposed to having to read the files into memory. The merge and sort functions will be discussed in the following section.

3 Complexity and Runtime

The primary operation that the program performs is a binary search through the records in the main database for the purpose of finding records. This is completed in $O(\log n)$ time. This can be done because the main database is sorted. A secondary search that is used when the record is not found in the main database searches the overflow file in $O(n)$ time. This is a poor sort because the overflow file cannot be guaranteed to be sorted and the process of sorting it would be $O(n \log n)$ time. However, since we can guarantee that the record size is small, this is not a problem.

When merging the overflow with the main database, the program invokes linux sort which uses a parallelized merge sort. This is completed in $\Theta(\frac{n \log n}{p})$ time where p is the number of processors used. Linux sort uses as many processors as the machine has available up to a maximum of 8, where diminishing returns begins - however, it can be modified to use more but this was not necessary for our program. Unfortunately, when databases get massive, the additional $\frac{1}{p}$ will not impact the runtime and it will be $O(n \log n)$ time anyway. The way we invoked the linux call requires only one use of the $\Theta(\frac{n \log n}{p})$ function because we can exploit the ability to handle multiple files. It could have been completed in the same actual time, but double the asymptotic time, by calling sort on just the overflow file and merging the two files after. This can be done because we can guarantee that the database file will be sorted, so we do not need to call sort on it. The function has a *--merge* option that allows just a merge to be performed, however this is still in $\Theta(\frac{n \log n}{p})$ time. One note about the linux sort is that it requires the *-n* or the *--numeric-sort* flag in order to properly sort our database. This is because it contains a header line which is entirely made up of alpha characters whilst the rest of the database begins each record with digit characters. The default sort places the digits above the header, which had to be fixed by adding the *-n* option.

When opening an existing database, the binary search requires that we know how many records are present. This can be done with a simple $O(n)$ run through the file to count, but it can actually be done in constant time, $\Theta(1)$ with the use of the operating system. C++ method fstream contains a function called *tellg* and *seekg* which tell and move the get pointer of fstream objects. This can be utilized to find the last element thanks to the $ios :: end$ flag, which can

be manipulated to give us the number of lines in the file thanks to the fixed record size.

```
1  getRecordCount(fstream infile)
2  {
3          infile.seekg(0, ios::end);
4          long position = file.tellg();
5          NUM_RECORDS = (position - RECORD_SIZE) / RECORD_SIZE;
6  }
```

This allows us to easily accept any incoming database without any problems handling the number of records contained.

The program does contain the given GetRecord function which will get a record in constant time, however, this requires that the database be sorted and consecutive. This cannot be guaranteed, and is not worth the resources to determine if it is consecutive or not. For this reason, it is not used in the program.

No other out-of-the-ordinary functions or algorithms were used in this assignment other than the above discussed. The program works as intended and required. We have attached a typescript as well as the files that were generated and modified during the typescript run - they are labeled as such.