**CSCE 4523 Database Management Systems**

**Homework 1**

**Due: Monday, February 3 at 11:59pm**

**By: Tyler Gerth and Gillian Dirkson**

**Objectives**

The objective of this homework was to create a text-based database of fixed-length fields. A user would be able to use this application to perform database operations such as creating a database, opening and closing a database, adding and deleting records, displaying and updating records, and generating a report of the top 10 records in the database. Since the goal of this project was to simulate a database, all these records would be stored on disk in a file, and whenever operations occurred, only one or two records would be brought into main memory at a time. The three main files to be used were the data file, which stored all the records in the database, the config file, detailing metadata about the database, like the size of the fixed-length fields, and an overflow file to store new added records until they were merged and sorted with the existing database records.

**Approach**

In order to implement this database, we used the Java programming language due to our familiarity with it, as well as for its class structure, which would allow for an easier design of the methods and variables needed for the database. We decided to split the structure of the database into a few separate files. The first was the application wrapper to loop until the user quit the application, allowing one to continue performing database operations. The second main one was a class for creating a new database. This class would take in and parse a csv file, adding each line as a separate record in the database. Additionally, it would create the config and overflow files for that database, so the metadata could be stored and the database could persist even if it was closed and reopened. Finally, the last main class was the database operations class, which contained functions for all the operations, such as adding, deleting, and displaying, needed for a database.

We chose to make each record the combination of 6 fixed-length fields – rank (5 bytes), company name (40 bytes), city (20 bytes), state (3 bytes), zip code (6 bytes), and number of employees (10 bytes). This meant that each line would be a total of 85 bytes on Linux and 86 bytes on Windows once new line characters were taken into account. An example record can be seen below in Figure 1. Since the fields were fixed length and could be easily obtained by taking reading a substring of the record, we chose to not use any delimiters between them. The primary key that the records were sorted by was the company name, since this is the most likely field that a user would know in advance when wanting to find out more information about Fortune 500 companies.



Figure 1: Example of a record in the database

We used the config file to store metadata about each database that the user created. This would include things like the size of the fixed length fields, total number of records in the normal database and the overflow file, and the highest rank currently set for a record. The config file can be seen below in Figure 2. As for the uses of the config file, the number of overflow records would get incremented each time a user added a record and then reset to 0 once five records were added, since they would be merged and sorted back into the normal database at this point. The highest rank was also incremented with each added record so as to help make sure two records would not have the same rank.



Figure 2: Example of a database’s config file

The sample code we used was for getting a record and performing a binary search to see if a company name existed in the database. This code was provided by Dr. Gauch along with the handout for this homework.

The overflow file was handled by having records added to it when a user added a new record to the database. Once five records had been added, all the records in the overflow file were merged back into the normal database by determining where they fit in alphabetically by the company name. To find this location, we used a binary search and then shifted the rest of the normal database down one position. In other functions, such as deleting, updating, and displaying records, the normal database was first checked with a binary search to see if the company name inputted existed in it, and if not, the overflow file was linearly searched. Finally, we added functions to get the number of records in the overflow file (to see when it grew to over 4 records) and get a record from the overflow file (to move it into the normal database when merging occurred or it needed to be displayed, updated, or deleted).

**Results**

Most of the error handling in this program was dealing with file and user input, such as making sure the user entered a valid database name when opening a database or truncating or expanding input that was too long or short respectively for the fixed length records. The overall application ended up being fairly efficient, since only one record from the database would be brought into memory at a time, meaning that main memory wasn’t being used up too much. Another thing that helped was by using binary search rather than linear search for any operation that required a specific record. This was possible because the database was sorted by its primary key, the company name, so the binary search could recursively split the database in half over and over until a record was found. If one was not found however, then the overflow file, which was not sorted, would be linearly searched (which was why it was kept to such a small size).

One part of the program that might’ve been able to be more efficient was the ‘Delete Record’ operation. The initial idea was to set the company name of whichever record the user wanted to delete to ‘MISSING\_RECORD’ and then remove it when merging in added records. However, doing this would cause the database to no longer be sorted, since a company name like Walmart could be changed to MISSING\_RECORD, and thus binary search would no longer work properly. To handle this problem, we decided to just remove the record altogether when the user requested it, by bringing each record in the database into main memory one at a time and then transferring them over to a new file. If a MISSING\_RECORD name was found, it was skipped. Finally, the original database file would be deleted and the new file would be set as the new database. Though this solution fixed the problem, it meant that the database was now being linearly searched when deleting records, which is not very performant.

For our group, we decided to split up the database operations and each work on a few so that we wouldn’t have overlapping effort. Gillian worked on displaying records, updating records, creating the report of the top 10 records, and testing the program with test runs. Tyler worked on creating the database, opening and closing it, adding and deleting records, and quitting the program. We managed the project by using Git to both create branches, implement our functionality, and then review each other’s code before it was merged back into master.

**Testing**

For testing we followed the test cases specified in the test cases document. After that a second script was created containing extra test cases. This demonstrated how records in the overflow file could be displayed, updated and deleted. It also demonstrated how a new database cannot be opened until the current one is closed. Prior to running a script, the program was fully tested on turing. All functions were looked over, and any that did not perform correctly were updated to work within turing. In the end the program performed as expected and passed all test cases. One thing that will be different is the fact that after a new record is added it can be displayed. In the test case I believe this is something that would be expected to return “NOT FOUND” however, we added the functionalty, so that those in overflow could still be shown.

**Typescript**

Provided in two additional files.