CSc 460 — Database Design
Spring 2022 (McCann)

http://u.arizona.edu/~mccann/classes/460

Program #2: Extendible Hashing

Due Date: February 9th, 2022, at the beginning of class

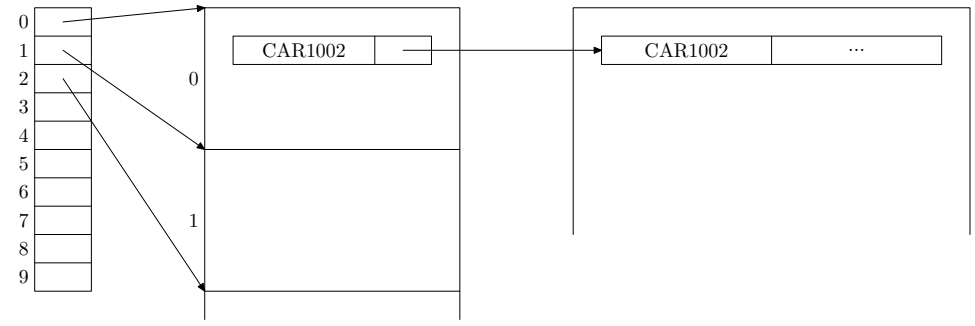
Special Option:
You MAY choose to do this assignment with a partner (i.e., in teams of two).
See the "To Partner or Not to Partner" section, below, for details.

Overview: In class we recently talked about Extendible Hashing under the assumption that the directory is indexed on bit patterns derived from the search key. This isn't a law; if the situation permits, we can index based on any part of the key we wish, even full characters.

Assignment: Write a complete, correct, and well–documented Java 16 program named `Prog2.java` that (first task) creates an extendible hashing index for the `Offsets-Database.bin` file you created for Program #1 and (second task) uses this index to satisfy some simple queries.

First Task: Build the Index. For this assignment, you will be using `Project ID` (a.k.a. the first field) as the key for the index. In class, we demonstrated Extendible Hashing using binary key values (0s and 1s). We can do it almost as easily with decimal values (0 — 9). Unfortunately, our Project IDs are a mixture of letters and digits, so we need a scheme to convert something like 'CAR1002' to an integer. Here's our scheme: Access the characters of the key right–to–left, get the least–significant digit of the ASCII value of each character, and concatenate them to form a string of digits. For example, when read right–to–left, 'CAR1002' is '2001RAC'. The ASCII values of those characters are 50, 48, 48, 49, 82, 65, and 67. Their least–significant digits are 0, 8, 8, 9, 2, 5, and 7. Concatenating them produces the string '0889257', which we will use to store or access CAR1002's record in our index.

Because we're using 10 digits (0–9) instead of two (0 and 1), our Extendible Hashing directory's sizes will be powers of 10 instead of powers of two. We can easily index into our Extendible Hash directory with full digits rather than binary bits by starting the directory with 10 pointers (labeled 0 through 9) instead of two, as demonstrated below:



As shown in this illustration, the index record for the key 'CAR1002' is stored in the first bucket of the index. Recall that the digit string we computed for 'CAR1002' was '0889257'. It starts with a '0', so we access the pointer stored in the directory at index 0. As it happens, that pointer is referencing the first bucket of the index file. That bucket has available space for the index record, so that's where we would store CAR1002's index record. We know where CAR1002's database file record is located because we're reading the records one at a time from the database file to build the index.

(Continued . . .)

The directory is holding "pointers" to the buckets in the Hash Bucket File. Initially, the bucket file is empty; a bucket is added when the first index entry needs to be stored. For this program, size a bucket so that it can hold a maximum of 50 entries, where each entry is a pair of values: a Project ID string and a "pointer" into the Database File (the binary file you created in Program #1). These "pointers" are actually just byte offsets into the files. The offset of the first bucket is 0. The offset of the n^{th} bucket is n times the size of a bucket. A similar calculation gives the "pointer" to a record in the database file.

What's in a bucket? A bucket is effectively an array of 'slots' (each holding a key–pointer pair) and perhaps a count of how many of those slots are currently in use. (Note that this count could also be held in the directory entries if desired.) Recall that when a bucket is full and a new addition arrives, we split the bucket, and perhaps 'double'[1] the directory as well. Creating the hash bucket file as a binary file makes the splits easier to handle.

Second task: Use the Index. Prompt the user to enter an Project ID value suffix (for example, '1', 'S1', and 'CS1' are all suffixes of 'VCS1') and return to the user the `Project ID`, `Project Name`, and `Total Credits Issued` fields of the binary database file records associated with all keys having the given suffix, **and** a count of the number of data records returned. Naturally, you are expected to use the extendible hash index to find all matching records, rather than directly searching the database file.

Data: The input data that this program needs to construct the index is the binary file created from `Offsets-Database.csv` by your Program #1. As mentioned above, the index will be constructed on the Project ID field of this binary file, using the conversion to digits scheme described above.

For the second part of the assignment (the queries), we will not be supplying sample queries; we expect that you can invent your own suffixes and perform your own testing.

Output: Display to the user the `Project ID`, `Project Name`, and `Total Credits Issued` field values of the data records associated with all of the keys having the given suffix, followed by the count of the number of records found that have that suffix. Use the same basic output format you used in Program #1, ending with the line "# records matched your query."

Hand In: Submit your completed program file(s) using the `turnin` facility on lectura. The submission folder is `cs460p2`. Instructions are available from the document of turnin instructions linked to the class web page. Be sure to name your main program source file `Prog2.java` so that we don't have to guess which files to compile, but split up your code over additional files as appropriate to achieve reasonable modularity.

Remember: Be sure that your program works successfully on lectura before you submit it.

Other Requirements and Hints:

- Start early! This is not a trivial program; it will take significant time to complete.
- Work modularly. For example, you will undoubtedly want to create an extendible hash index class.
- When you test your code, initially use only a subset of your binary file and small index buckets. If everything seems OK, then try it on larger subsets and buckets, preferably selected to test specific situations. Only then try your program on the complete file and full–size buckets.
- Remember to comment your code according to the style guidelines. In particular, please use the block commenting templates to ensure that you have all of the information we require.

(Continued . . .)

[1] Because our directory is Base 10 instead of Base 2, the correct word is "decuple" (10x) instead of "double" (2x). If we were indexing on the 26 upper-case letters, the term would be "sexvigintuple". Seriously!

To Partner or Not to Partner: As mentioned at the top of this handout, you have the option to work on this assignment in teams of two class members. Before you rush around the room trying to secure your favorite "code monkey" as a partner, read on.

If you work with a partner:

1. Include both names in your program documentation (so that you both get credit!). Each team of two should submit the program just once.
2. Your team will have to store the directory *within* the hash bucket file. It cannot be in a separate file.
3. Your team must write a separate program (named `Prog2q.java`) to perform querying. This second program will have to begin by loading the directory into memory before processing any queries.
4. The index creation portion of the program is to conclude by displaying the following information about the index: The final global depth of the directory, the number of unique bucket pointer values held in the final directory, the number of buckets in the hash bucket file, and the average bucket occupancy (as a number of entries).
5. If your team needs to use late days, each team member will lose a late day for every day the assignment submission is late. For example, if partners Bob and Frank turn in their program two days late, Bob loses two late days and Frank loses two late days.

If you work solo:

- You **do not** need to store the directory into a file at all (you can just hold it in memory), you **do not** need to print any statistics, and you should write just **one** program (`Prog2.java`). This means that your index creation program is also your querying program: Create the index, then immediately begin getting queries from the user and answering them (by accessing the on–disk index, of course).

Obviously, the point of the additional chores is to balance the workload a little more equitably. Even so, I think that (personality and scheduling conflicts aside) this assignment will be easier to do as a team. But, the choice is yours.