CSc 560 — Database Systems Implementation
Fall 2022 (McCann)

http://u.arizona.edu/~mccann/classes/560

Program #1: Variable–Length Record Storage

(100 points)

Due Date: September 22nd, 2022, *at the beginning of class*

Overview: We recently learned how variable–length records can be stored in a DBMS, under the assumption that records are shorter that the block size. In this assignment, you will implement a version of that storage scheme, use it to store data from an online data set, and will display statistics about the storage scheme and results from a simple form of query.

Assignment: The parts of the assignment are: (1) create an ASCII text DB file named `dbfile.txt` containing data that is stored as variable–length records within virtual fixed–length blocks; (2) report some basic statistics about the newly–created DB file; and (3) find records within the DB file based on a search key range and display the content of those records.

Using your choice of programming language[1], write two complete, well–documented programs named `Prog1A` and `Prog1B` (capitalized, and with file extensions, appropriate for your choice of language) that handle all three of the parts. `Prog1A` handles part 1, while `Prog1B` handles parts 2 and 3. The details of each part follow:

Part 1: The first task for your Prog1A program is to read the provided data and insert it into blocks stored within `dbfile.txt`, which will mimic a multi–block disk file. Available on `lectura` is a comma–separated–value (CSV) source file containing some data about a few thousand CPUs and GPUs, specifically product names, types, dates, and quantities of transistors. The middle two field values are fixed–size strings; the others are variable-length strings (yes, we're treating integers as strings for this assignment). See the "Data" section, below, for the path to the CSV file as well as content examples.

Your DB file of blocks of variable–length records will be represented within a text file consisting of one–byte ASCII characters. Everything (data and metadata) is stored in the DB file using ASCII character sequences (one byte per character), even numeric values. This keeps the data consistent, and likely makes debugging easier (because you can use a text editor to examine at the content of your DB file).

The block size is 1,000 bytes. The data and metadata stored within a block will follow the ideas explained by the textbook in sections 13.2.2 (pages 592-4) and 13.3.1 (pages 596-7), modified as follows.

Record Structure: You are to use the variable–length record structure from Figure 13.5 (page 592), with the following changes. First, the offset and length values of the variable–length fields are all two–digit integers. This will work because our data is fairly compact. Second, our null value 'bitmap' is four characters, with '0' for "not null" and '1' for "null," stored immediately after the list of offset/length pairs and immediately before the fixed–length string values. This ordering places the metadata at the front of the record and the data at the back.

For example, imagine this set of abbreviated field values: A1, CPU, 2018, 51.[2] We have two variable–length strings (the name and the # of transistors), so we need two pairs of offset/length values. We also need four bytes for the null 'bitmap', three for "CPU", and four for "2018", plus in this example two each for "A1" and "51". In sequence, our record content requires 23 bytes and looks like this:

(Continued...)

[1]See the "Other Requirements and Hints" section for practical limitations on your choice.
[2]The real chip names are longer, and the actual data's dates are 10 characters in size, not four.

```
         Metadata                Data
   |-----------------------|---------------------|

    1 9 0 2 2 1 0 2 0 0 0 0 C P U 2 0 1 8 A 1 5 1   <--- Content
    - - - - - - - - - - - - - - - - - - - - - - -
    0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 2 2 2   <--+ Byte Count
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2   <--+
```

The initial "19" and "02" pair tells us that 'A1' is stored starting at byte 19 and has a length of 2 bytes. We know where to find the offsets, lengths, bitmap metadata, and fixed–length strings because we have a pre-determined record structure and field descriptions.

Storage of missing values varies depending on the type. A fixed–length string value that is missing still needs to have something stored in its locations within the record, to maintain the positioning of the following fixed–length string(s), if any. The bitmap tells us to ignore those characters. Missing variable–length strings don't need placeholders for the missing data, but do for the offset and length. Again, the bitmap tells us that they can be ignored.

Block Structure: For our block structure, we will follow the slotted page structure shown in Figure 13.6 on page 593. Because blocks are bigger than records, we will use three–digit integer sizes and locations for record metadata (the record offsets and lengths). For consistency, we'll also use three–digit values for the # of entries in the block and the location of the last byte of the free space.

For example, here's a reduced block of just 100 bytes (0 – 99), containing two records. The first one added to the block is the example record shown above. You should be able to determine the content of the second on your own:

```
002051077023052025...........free.space.............190322030000GPU2020C9X109190221020000CPU2018A151
----------------------------------------------------------------------------------------------------
0000000000111111111122222222223333333333444444444455555555556666666666777777777788888888889999999999
0123456789012345678901234567890123456789012345678901234567890123456789012345678901234567890123456789
```

Create your initial DB file with a single 'empty' block. An empty block contains a record quantity of zero and the free space ends with the last byte. You may store whatever filler you like within the free space bytes.

Free–Space Map: When a new record needs to be inserted, your program is to check for available space of sufficient size in existing blocks using the free–space map scheme described in section 13.3.1 on page 596. You are to use a four–bit (max 16) integer value scheme (rather than the three-bit version used in the example), and a single–level map that is held and maintained in memory. If multiple blocks have sufficient free space, choose the block closest to the start of the DB file. When a new block is needed to hold a new record, append a new empty block to the DB file and add it to the free–space map. Do not pre–allocate blocks for anticipated future needs, apart from the very first block.

Conclude Prog1A by outputting the content of the free–space map as a labeled linear table of integer values.

<u>Part 2</u>: Prog1B is to start by retrieving from the DB file, and displaying, the following statistics. Create your own output format for this information. Label all information clearly!

1. The total quantities of blocks and records contained within the DB file.
2. For each block, display its quantity of free space bytes.
3. The field values of the records held by the first block, displayed in insertion order (that is, the first record displayed is the first record inserted)
4. Same as (3), but for the records within the last block.

(Continued...)

Note that, for a small DB file, the first block may also be the last block. In that case, display its content twice — once as the first block, and once as the last.

Part 3: As additional confirmation that your DB file was correctly constructed, conclude Prog1B by allowing the user to query its content to answer questions of this form: "Display all field values of the records of the chips that have at least M, but no more than N, transistors." Prompt the user for the values of M and N, and display the results in any order. Continue prompting the user for M and N value pairs and displaying results until either or both values are -1.

Data: Write your Prog1A to accept the complete CSV source file pathname as a command–line argument. The complete `lectura` pathname of our data file is `/home/cs560/fall22/chips.csv`. Your program can read the file directly from that directory; just provide that path when you run your program. (There's no reason to waste disk space by making a copy of the file in your CS account.) Of course, if you're developing your code on your own computer, you'll need to download a copy.

Write your Prog1B program to accept the complete path of `dbfile.txt` on the command line.

Each of the lines in the `.csv` file contains four fields (columns) of information, in this order: The product name, its type ('CPU' or 'GPU'), its release date, and the approximate quantity of transistors it contains (in millions). We will represent all four values as strings of ASCII characters. The type and the date are fixed–length (three and 10 characters, respectively). The name and quantity are of varying lengths. Here is a sample of how data appears in the CSV source file, including the line of field names that appears at the start of the file:

```
Product,Type,Release Date,Transistors (million)
AMD C-70,CPU,2012-09-01,
Intel A110,CPU,2007-06-01,176
```

This sample demonstrates some of the data situations that your program(s) will need to handle:

- Because the first line (the field names) contains only metadata, that line **must not** be stored in the record file. Code your program to ignore that line.
- In some records, some field values are missing. For example, in the first data line, notice the trailing comma, which indicates that the quantity of transistors is missing. Happily, we know how to handle this in our DB file: Set the characters of the record's null bitmap appropriately.
- Be aware that I have not combed through the data to see that it is all formatted perfectly. No, I'm not (that) lazy. I didn't comb through it because corrupt and oddly–formatted data is a huge headache in data management and processing. I'd be happy to hear that this file holds a surprise or two, because you'd get to think about how to deal with surprise data issues you may find in the CSV file, and to ask me questions about them (preferably publicly in class or on Piazza, so that my responses can be heard/read by the whole class) as necessary.

Output: Basic output details for the program are stated in the Assignment section, above. Please ask (again, preferably in public posts on Piazza) if you need additional details.

Hand In: You are required to submit your completed Prog1A and Prog1B program files using the `turnin` facility on lectura. The submission folder is `cs560p1`. Instructions are available from the document of submission instructions linked to the class web page. In particular, because I will be grading your program on lectura, it needs to run on lectura, so be sure to test it on lectura. Feel free to split up your code over additional files if doing so is appropriate to achieve good code modularity. Submit all files as–is, *without* packaging them into `.zip`, `.jar`, `.tar`, etc., files.

(Continued...)

Want to Learn More?

- https://chip-dataset.vercel.app/ ⟸ This site is the source of our data. The full data file has several more fields than we are using. You don't need to visit this page; I'm providing it in case you're interested in learning more about the reason the data exists.

Other Requirements and Hints:

- As mentioned at the start of this handout, you can develop your code in any language you wish, subject to one very practical restriction: The language must be installed for general use on lectura. Java 16, gcc 9.4 (for C and C++), Python 3.8.10, and several others are available. If you really want a language that isn't currently on lectura, email `lab@cs.arizona.edu` and request that it be installed; they might do it, they might not.

  I will be testing your programs on lectura, which is why you're limited to what it offers. Be sure to include in your 'external' block commenting the commands to use to successfully compile and execute your code on lectura.

- Don't know how to do something in your language of choice? You may consult the internet, textbooks, friends, etc., for such information. Learning from other people and sources how something may be accomplished is not a violation of academic integrity. Finding a way to get someone else to apply that approach to the construction of your code, knowingly or otherwise, is a violation, and will be sanctioned appropriately. Write your own code!

  Something non–obvious that is also not a violation: You can share output with each other to confirm that your program(s) are working correctly. You can even share `dbfile.txt` files if you want to do so; that's also output. Don't share your code.

- Don't "hard–code" values in your code if you can avoid it. For example, don't assume a certain quantity of records in the input file. Your code should automatically adapt to simple changes, such as more or fewer lines in a file or changes to the file names or file locations. Another example: I am likely to test your program with an input file of just a few records, perhaps even no records. I expect that your program will handle such situations gracefully. The characteristics of the fields in the `.csv` file (types, order, field separators, etc.) will not change.

- Are you used to writing your code documentation only after the code is working (or, worse, not writing any documentation at all)? I encourage you to give this a try: Comment your code according to the style guidelines *as you write the code* (not two hours before the due date and time!). Explaining to yourself in words what your code must accomplish *before* you write that code is likely to result in better code sooner. My documentation requirements and some examples are available here: http://u.arizona.edu/~mccann/style.html

- You can make debugging easier by using only a few lines of data from the data file for your initial testing. Try running the program on the complete file only when you can process a few reduced data files. The data file is a plain text file; you can view it, and create new ones, with any text editor. I'll be doing exactly that to create test cases for grading. (And, yes, you can share your testing CSV files with each other!)

- Remember that you have 'late days' can be used to submit the program after the due date without a point penalty. Of course, it's best if you don't use any late days at all; you may need them for a later assignment. The best way to not use late days is to ...

- **Start early!** There's plenty for you to do here; give yourself time to do it, and do it well.