



## CSE 333 13su Homework #3

**out:** Friday, July 26, 2013  
**due:** Thursday, August 8, 2013 by 11:00 pm.  
[ [summary](#) | [part a](#) | [part b](#) | [part c](#) | [bonus](#) | [how to submit](#) | [grading](#) ]

### Summary

For homework #3, you will build on your homework #2 implementation. In Part A, you will write code that takes an in-memory inverted index produced by hw2 and writes it out to disk in an architecture-neutral format. In Part B, you will write C++ code that walks through an on-disk index to service a lookup. Finally, in Part C, you will write a query processor that serves queries from multiple on-disk indices.

As before, please read through this entire document before beginning the assignment, and please start early! There is even more code involved with HW3 than in earlier homeworks, and since this is your first serious attempt to use C++, you should expect to encounter a lot of problems along the way. Also, manipulating on-disk data is trickier than in-memory data structures, so plan for some time to get this part right.

In HW3, as with HW2, you don't need to worry about propagating errors back to callers in all situations. You will use `assert()`'s to spot errors and cause your program to crash out. We will not be using C++ exceptions in HW3.

### Part A -- finish our memory-to-file index marshaller.

#### Context.

Keeping a search engine index in memory is problematic, since memory is expensive and also volatile. So, in Part B, you're going to write some C++ code that takes advantage of your HW2 implementation to first build an in-memory index of a file subtree, and then it will write that index into an **index file** in an architecture-neutral format.

What do we mean by architecture-neutral? Every time we need to store an integer in the file's data structure, we will store it in **big endian** representation. This is the representation that is conventionally used for portability, but the bad news is that this is the opposite representation than most computers you use: x86 computers are little endian. So, you will need to convert integers (whether 16-bit, 32-bit, or 64-bit) into big endian before writing them into the file. We provide you with helper functions to do this.

The good news is that we're going to keep roughly the same data structure inside the file as you built up in memory: we'll have chained hash tables that are arrays of buckets containing linked lists. And, our inverted index will be a hash table containing a bunch of embedded hash tables. But, we need to be very precise about the specific layout of these data structures within the file. So, let's first walk through our specification of an index file's format. We'll do this first at a high level of abstraction, showing the major components within the index file. Then, we'll zoom into these components, showing additional details about each.

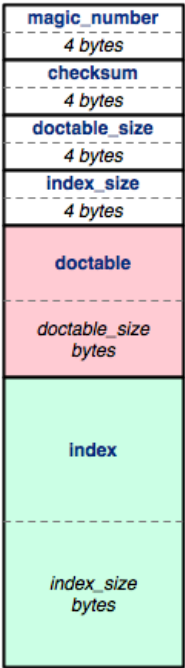
At a high-level, the index file looks like the figure on the right. The index file is split into three major pieces: a header, the doctable, and the index. We'll talk about each in turn.

**Header:** an index file's header contains metadata about the rest of the index file.

The first four bytes of the header are a [magic number](#), or format indicator. Specifically, we use the 32-bit number **0xCAFEF00D**. We will always write the magic number out as the last step in preparing an index file. This way, if your program crashes partway through writing one, the magic number will be missing, and it will be easy to tell that the index file is corrupt.

The next four bytes are a [checksum](#) of the *doctable* and *index* regions of the file. A checksum is a mathematical signature of a bunch of data, kind of like a hash value. By including a checksum of most of the index file within the header, we can tell if the index file has been corrupted, such as by a disk error. If the checksum stored in the header doesn't match what we recalculate when opening an index file, we know the file is corrupt and we can discard it.

The next four bytes store the size of the *doctable* region of the file. The size is stored as a 32-bit, unsigned, big endian



index file

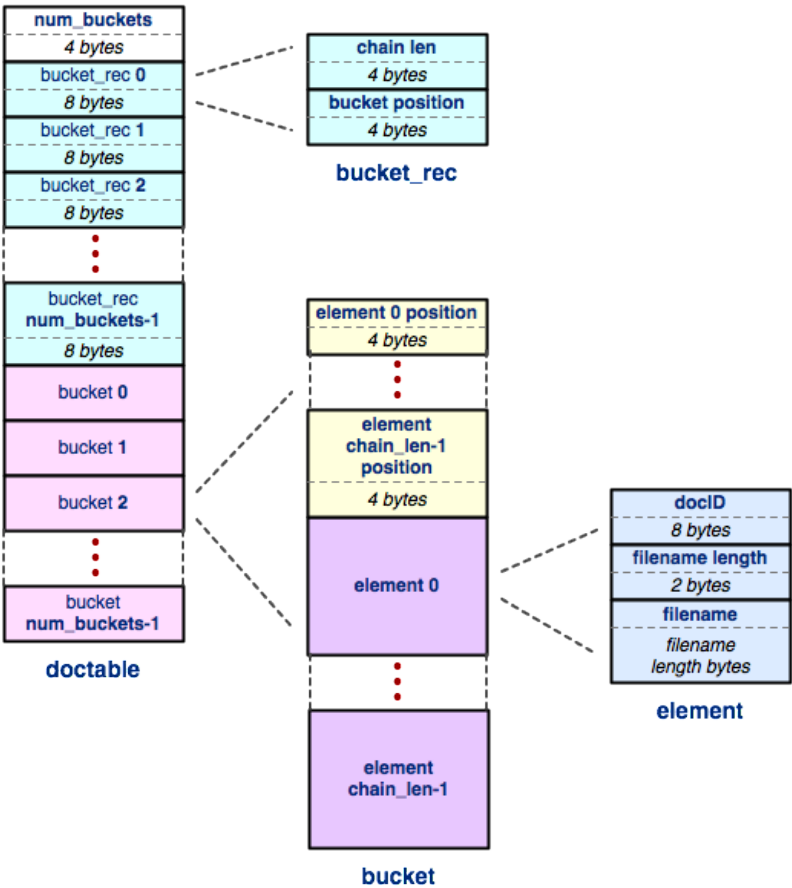
integer.

The final four bytes of the header store the size of the *index* region of the file, in exactly the same way.

**Doctable:** Let's drill down into the next level of detail by examining the content of the *doctable* region of the file. The doctable is a hash table that stores a mapping from 64-bit document ID to an ASCII string representing the document's filename. This is the docid\_to\_docname HashTable that you built up in HW2, but stored in the file rather than in memory.

The doctable consists of three regions; let's walk through them, and then drill down into some details.

- num\_buckets:** this region is the simplest; it is just an unsigned, 32-bit big endian integer that represents the number of buckets inside the hash table, exactly like you stored in your HashTable's HashTableRecord.
- an array of bucket\_rec records:** this region contains one record for each bucket in the hash table. A bucket\_rec record is 8 bytes long, and it consists of two four-byte fields. The *chain len* field is a four byte integer that tells you the number of elements in the bucket's chain. (This number might be zero if there are no elements in that chain!) The *bucket position* field is a four byte integer tells you the offset of the bucket data (i.e., the chain of bucket elements) within the index file. The offset is just like a pointer in memory, or an index of an array, except it points within the index file. So, for example, an offset of 0 would indicate the first byte of the file, an offset of 10 would indicate the 11th byte of the file, and so on.
- an array of buckets:** this region contains one bucket for each bucket in the hash table. A bucket is slightly more complex; it is a little embedded data structure. Specifically, each bucket contains:
  - an array of element positions:** since elements are variable-sized, rather than fixed-sized, we need to know where each element of the bucket lives inside the bucket. For each element, we store a four-byte integer of containing the position (i.e., offset) of the element within the index file.
  - an array of elements:** at each position specified in the element positions array lives an element. Since this is the docid-to-filename hash table, an element contains a 64-bit document ID and a filename. The document ID is an unsigned, big endian integer, like everything else. Next, we store a 16-bit (2 byte) unsigned, big endian integer that contains the number of characters in the file name. Finally, we store the filename characters (each character is a single ASCII byte). Note that we do NOT store a null-terminator at the end of the filename; since we have the filename length in an earlier field, we don't need it!

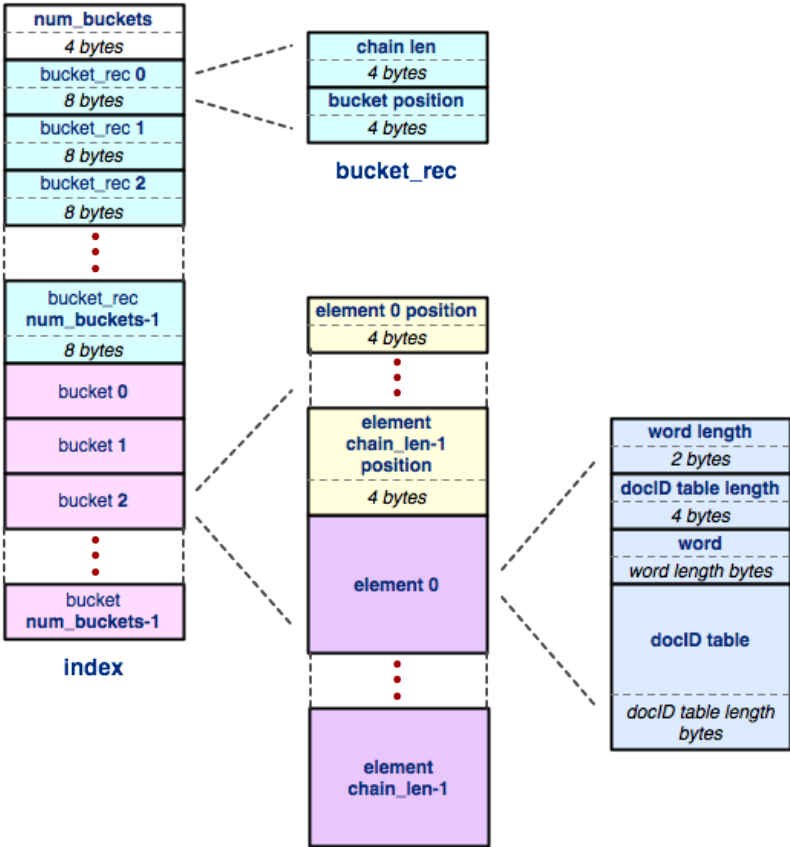


Phew! That wasn't so bad.

**Index:** The index is the most complicated of the three regions within the index file. The great news is that it has pretty much the same structure as the doctable: it is just a hash table, laid out exactly the same way. The only part of the index that differs from the doctable is the structure of each element. Let's focus on that.

An index maps from a word to an embedded docID hash table, or *docID table*. So, each element of the index contains enough information to store all of that. Specifically, an index table element contains:

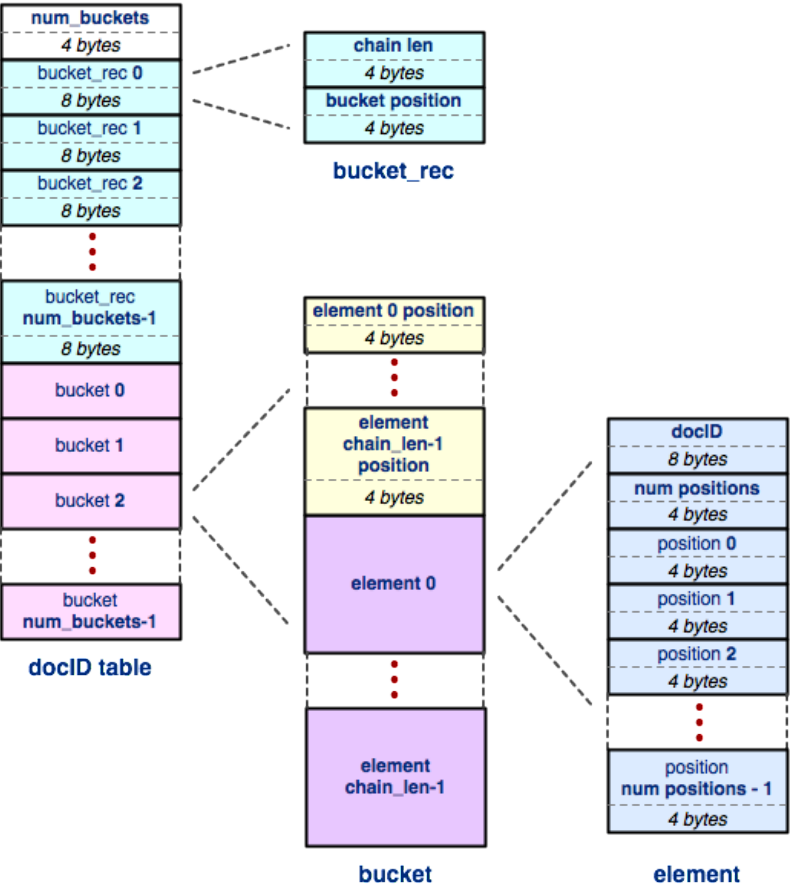
- a two-byte unsigned integer that specifies the number of characters in the word.
- a four-byte unsigned integer that specifies the number of bytes in the embedded docID table.



- an array of ASCII characters that represents the word; as before, we don't store a NULL terminator at the end.
- finally, the element contains some variable number of bytes that represents the docID table. So, all we need to understand now is the format of the docID table. I'm sure it's format will come as no surprise at this point...

**docIDTable:** like the "doctable" table, each embedded "docIDTable" table within the index is just a hash table! A docIDTable maps from a 64-bit docID to a list of positions with that document that the word can be found in. So, each element of the docID table stores exactly that:

- a 64-bit (8-byte) unsigned integer that represents at docID.
- a 32-bit (4-byte) unsigned integer that indicates the number of word positions stored in this element.
- an array of 32-bit (4-byte) unsigned integers, sorted in ascending order, each one containing a position within the docID that the word appears in.



So, putting it all together, the entire index file contains a header, a doctable (a hash table that maps from docID to filename), and an index. The index is a hash table that maps from a word to an embedded doctable. The doctable is a hash table that maps from a document it to a list of word positions within that document.

**What to do.**

The bulk of the work in this homework is in this step. We'll tackle it in parts.

1. Change to the directory that has your hw1 and hw2 directories in it. Click (or right-click if needed) on this [hw3.tar.gz](#) link to download the

archive containing the starter code for hw3. Extract its contents (`tar xzf hw3.tar.gz`). You will need the hw1 and hw2 directories in the same folder as your new hw3 folder since hw3 links to files in those previous directories.

2. Look around inside of hw3/ to familiarize yourself with the structure. Note that there is a libhw1/ directory that contains a symlink to your libhw1.a and a libhw2/ directory that contains a symlink to your libhw2.a. You can replace your libraries with ours (from the appropriate solution\_binaries directories) if you prefer.
3. Next, run "make" to compile the three HW3 binaries. One of them is the usual unit test binary. Run it, and you'll see the unit tests fail, crash out, and you won't yet earn the automated grading points tallied by the test suite.
4. Now, take a look inside fileindexutil.h. This header file contains some useful utility routines and classes you'll take advantage of in the rest of the assignment. We've provided the full implementation in fileindexutil.cc. Next, look inside fileindexwriter.h; this header file declares the WriteIndex() function, which you will be implementing in this part of the assignment. Also, look inside buildfileindex.cc; this file makes use of fileindexwriter.h's WriteIndex(), and your HW2 CrawlFileTree(), to crawl a file subtree and write the resulting index out into an index file. Try running the solution\_binaries/buildfileindex program to build one or two index files for one or for a directory subtree, and then run the solution\_binaries/filesearchshell program to input.
5. Finally, it's time to get to work! Open up fileindexwriter.cc and take a look around inside. It looks complex, but all of the helper routines and major functions correspond pretty directly to our walkthrough of the data structures above. Start by reading through WriteIndex(); we've given you part of its implementation. Then, start recursively descending through all the functions it calls, and implement the missing pieces. (Look for MISSING: in the text to find what you need to implement.)
6. Once you think you have the writer working, compile and run the test\_suite as a first step. Next, use your buildfileindex binary to produce an index file (we suggest indexing ./test\_tree/enron\_email as a good test case). After that, use the solution\_binaries/filesearchshell program that we provide, passing it the name of the index file your buildfileindex produces, to see if it's able to successfully parse the file and issue queries against it. If not, you need to fix some bugs before you move on!

[Aside: If you write the index files to your personal directories on a CSE lab machine or onattu, you may find that the program runs very slowly. That's because home directories on those machines are actually on a file server, and buildfileindex does a huge number of small write operations, which can be quite slow over the network. To speed things up dramatically we suggest you write the files in /tmp, which is a directory on a local disk attached to each machine. Be sure to remove the files when you're done so the disk doesn't fill up.]

As an even more rigorous test, try running the "hw3fsck" program we've provided in solution\_binaries against the index that you've produced. hw3fsck scans through the entire index, checking every field inside the file for reasonableness. It tries to print out a helpful message if it spots some kind of problem.

Once you pass hw3fsck and once you're able to issue queries against your file indexes, then rerun your buildfileindex program under valgrind and make sure that you don't have any memory leaks or memory errors.

Congrats, you've passed part A of the assignment!

## Part B -- finish our index lookup code.

### Context.

Now that you have a working memory-to-file index writer, the next step is to implement code that knows how to read an index file and lookup query words and docids against it. We've given you the scaffolding of the implementation that does this, and you'll be finishing our implementation.

### What to do.

1. Start by looking inside FileIndexReader.h. Notice that we're now in full-blown C++ land; you'll be implementing constructors, manipulating member variables and functions, and so on. Next, open up FileIndexReader.cc. Your job is to finish the implementation of its constructor, which reads the header of the index file and stores various fields as private member variables. As above, look for "MISSING:" to figure out what you need to implement. When you're run, recompile and re-run the test suite. You won't yet pass all the tests for test\_fileindexwriter, but you'll get some points for it.
2. Next, you'll move on to HashTableReader.h. Read through it to see what the class does. Don't worry about the copy constructor and assignment operator details (though if you're curious, read through them to see what they're doing and why). This class serves as a base class for other subclasses. The job of a HashTableReader is to provide most of the generic hash-table lookup functionality; it knows how to look through buckets and chains, returning offsets to elements associated with a hash value. Open up HashTableReader.cc. Implement the MISSING components in the constructor and the LookupElementPositions function. When you're done, recompile and run the unit tests to see if you pass the hashtablereader unit test.
3. Now it's time to move on to DocTableReader.h. Read through it, and note that it is a subclass of HashTableReader. It inherits LookupElementPositions() and other aspects, but provides some new functionality. Next, open up DocTableReader.cc, and implement the MISSING functionality. See how well you do on its unit test when you're done. (By the way, you have been using valgrind to test for problems and memory leaks along the way, right? We definitely will when grading your assignment! ;)

4. Next, let's move on to `IndexTableReader.h`. Read through it and understand its role. Next, open up `IndexTableReader.cc` and implement the MISSING functionality. Test against the unit tests.
5. Next, do the same with `DocIDTableReader.h` and `DocIDTableReader.cc`. Note that the `TestQueryProcessor` unit test will hang in an infinite loop once you've finished this step; once you finish the next step that will get fixed.
6. We're almost there! Open up `QueryProcessor.h` and understand how it is supposed to work. Check out `test_queryprocessor.cc` for more information. Now open up `QueryProcessor.cc` and read through our implementation of the constructor and destructor. This part of the assignment is the most open-ended. We've given you the function definition for `ProcessQuery()`, and also a clue about what you should be building up and returning. But, we've given you nothing about its implementation. You get to implement it entirely on your own; you might want to define helper private member functions, you might want to define other structures to help along the way, etc.; it's entirely up to you. But, once you're finished, you'll need to pass our unit test to know you've done it correctly.

As a hint, you should be able to take inspiration from what you did to implement the query processor in HW2. Here, it's only a little bit more complicated. You want to process the query against each index, and then union each index's results together and do a final sort. (Use the STL sort support.) Remember that processing a query against an index means ensuring all query words are present in each matching document, and remember how ranking works. Then, once you have query results from each index, you'll append them all together to form your final query results.

As one more hint, once you think you have this working, move on to Part D and finish our `filesearchshell` implementation. You'll be able to test the output of your `filesearchshell` against ours (in `solution_binaries/`) as a final sanity check.

Also, now would be a great time to run `valgrind` over the unit tests to verify you have no memory leaks or memory errors.

You're done with part B!

### Part C - implement the search shell.

For Part C, your job is to implement a search shell, just like in HW2, but this time using your HW3 infrastructure you completed in parts A and B.

1. Open up `filesearchshell.cc` and read through it. Note that unlike parts A and B, we have given you almost nothing about the implementation of the `filesearchshell` besides a really long (and hopefully helpful) comment. Implement `filesearchshell.cc`.
2. Try using your `filesearchshell` binary. You can compare the output of your binary against [our transcript of using our solution](#). The transcripts should match precisely, except perhaps for the order of equally ranked matches.
3. Also, note that you can hit control-D to exit the `filesearchshell`. The `filesearchshell` ought to clean up all allocated memory before exiting. So, run your `filesearchshell` under `valgrind` to ensure there are no leaks or errors.

Congrats, you're done with (the mandatory parts) of HW3!!

### Bonus

There are three bonus tasks for this assignment. As before you can do none of them with no impact on your grade. Or, you can do one, two, or all three of them if you're feeling inspired!

1. **(Easy)**: You've probably noticed that the books seem to be consistently ranked higher than other document types in the corpus. This is for a pretty simple reason: books are long, so any given query word will, on average, appear more often in a book than in an email message, inflating the rank of the book. Solve this problem by normalizing the rank contribution of a word within a document by the document length; you'll have to modify the information calculated by hw2 and stored by hw3. In other words, instead of defining the rank contribution of a word in a document to be the word frequency, define it to be the  $(\text{word frequency}) / (\text{number of words in the document})$ . Do an informal study that evaluates whether this ranking is better or worse, and present your evidence.
2. **(Medium)**: Normalized term frequency is a better ranking function, but it also has a flaw: some words (e.g., "the" or "person") are inherently more frequently used than others (e.g., "leptospirosis" or "anisotropic"). There is a different ranking contribution function called "tf-idf", or term frequency inverse document frequency, that tries to compensate for this. Calculating tf-idf involves keeping track of how often a word appears across an entire corpus of documents, and normalizing term frequency within a document by the frequency across all documents. So, tf-idf measures how much more frequently than average a word appears in a given document. (Here's the [wikipedia page on tf-idf](#).) Implement tf-idf ranking. You'll have to populate a new hashtable for corpus word frequency and incorporate it into the index file format. Do an informal study that evaluates whether this ranking is better or worse than term frequency, and present your evidence.
3. **(Hard)**: use `valgrind` to do a performance analysis of hw3 query processing, and identify any major performance bottlenecks. If you find some obvious performance bottlenecks, attempt to optimize the code to reduce them. Present graphs that demonstrate the performance before and after your optimization, and the evidence you used to decide what to optimize.

## What to turn in

When you're ready to turn in your assignment, do the following:

1. In the hw3 directory, run "make clean" to clear out any object files and emacs detritus; what should be left are your source files.
2. In the hw3 directory:

```
bash$ make clean
bash$ cd ..
bash$ tar czf hw3_<username>.tar.gz hw3
bash$ # make sure the tar file has no compiler output files in it, but
bash$ # does have all your source and other files you intend to submit
bash$ tar tzf hw3_<username>.tar.gz
```

3. Turn in hw3\_<username>.tar.gz using the course dropbox linked on the main course webpage.

## Grading

We will be basing your grade on several elements:

- The degree to which your code passes the unit tests. If your code fails a test, we won't attempt to understand why: we're planning on just including the number of points that the test drivers print out.
- We have some additional unit tests that test a few additional cases that aren't in the supplied test drivers. We'll be checking to see if your code passes these.
- The quality of your code. We'll be judging this on several qualitative aspects, including whether you've sufficiently factored your code and whether there is any redundancy in your code that could be eliminated.
- The readability of your code. For this assignment, we don't have formal coding style guidelines that you must follow; instead, attempt to mimic the style of code that we've provided you. Aspects you should mimic are conventions you see for capitalization and naming of variables, functions, and arguments, the use of comments to document aspects of the code, and how code is indented. You can use [cpplint.py](#) to check your code for potential style issues. Use your judgment about how to deal with things it catches. You probably should fix most of the things it flags, but there may be some things that can be left alone.

Computer Science & Engineering University of Washington Box 352350 Seattle, WA 98195-2350 (206) 543-1695 voice, (206) 543-2969 FAX

[UW Privacy Policy](#) and [UW Site Use Agreement](#)