CECS 429 - Project Milestone 2

Milestone demo due June 21.

Overview

In this project milestone, you will extend your Milestone 1 to support a ranked retrieval system (in addition to the boolean retrieval functions you already programmed). You may work in teams of up to 3 people, but as you will see, more people in a group will necessitate additional work.

You will also incorporate a disk-based inverted index into your main application. This means your program will operate in two modes: in one mode, you will build a disk index for a specified directory, and in another mode you will process queries over an existing disk index.

When processing queries, you will also operate in two different modes: **boolean retrieval** mode, in which queries are treated as boolean equations, and only documents satisfying the entire equation are returned; and **ranked retrieval** mode, where the top **10** documents that satisfy a given "bag of words" query are selected and presented to the user. The user can select this mode at the beginning of the program.

Requirements

These **mandatory** requirements deal with building the index data structure and supporting ranked queries:

Building the index: Your `PositionalInvertedIndex` must be written to disk using the `DiskIndexWriter` class from Homework 3. Your index file must be constructed in the following pattern: $\mathrm{df}_\mathrm{t}\ d\ \mathrm{tf}_{t,d}\ p_1\ p_2\ \cdots\ p_i$, where d is the document ID, $\mathrm{tf}_{t,d}$ is the term frequency of a term in the document (i.e., the number of positions that the term appears at), and $p_1\ p_2\ \cdots\ p_i$ are each of the i positions of that term in that document. All document IDs and positions must be written as **gaps**.

Reading the index: You must create a new implementation of the `Index` interface, `DiskPositionalIndex`, which reads postings from an on-disk index as created by your `DiskIndexWriter`. Each `Index` interface method must be implemented; to program `getPostings(String term)`, you must locate the byte position within the `postings.bin` file for the given term. This information should be written to a B+ tree / relational database using a library of your choice; modify `DiskIndexWriter` so that it creates this data. Once you have a byte location, use a `seek` method on a `DataInputStream` (Java), `BinaryReader` (C#), or `File` object (Python) to jump to that position and then read and translate integer values from disk into `Postings` objects in your program.

Calculating document weights: Ranked retrieval requires calculating a "weight" of each document to use in *normalization*, so that long documents don't receive "extra relevance" just because they are longer. The weight of each document is called L_d and can be calculated during the indexing process. Each L_d value is equal to the *Euclidian normalization* of the vector of $\mathrm{w}_{d,t}$ weights for the document, where

$$\mathrm{w}_{d,t} = 1 + \ln\left(\mathrm{tf}_{t,d}\right)$$

Note that this formula only needs $\mathrm{tf}_{t,d}$, which is the number of times a particular term occurs in the document.

So to calculate L_d for a document that you just finished indexing, you need to know each term that occurred at least once in the document along with the number of times that term occurred; a `HashMap` from term to integer, updated as you read each token, can help track this. For each term in the final map, the integer it maps to is $\mathrm{tf}_{t,d}$ for that term and can be used to calculate $\mathrm{w}_{d,t}$ for that term. To find L_d, sum the squares of all the $\mathrm{w}_{d,t}$ terms, and then take the square root: $L_d = \sqrt{\sum_t \left(\mathrm{w}_{d,t}\right)^2}$.

Each L_d term needs to be written as an 8-byte `double` in document order to a file called `docWeights.bin`. Modify `DiskIndexWriter` so that it creates this file during the indexing process. Modify `DiskPositionalIndex` so it knows how to open this file and skip to an appropriate location to read a 8-byte `double` for L_d. L_d values will be used when calculating ranked retrieval scores.

Querying the index: You must not read the entire index into main memory when processing user queries. You must only read postings lists for the terms required to satisfy the query. In other words, you will not be using `PositionalInvertedIndex` when processing ranked or Boolean queries; that will fall to your new `DiskPositionalIndex`.

Your `DiskPositionalIndex` class **must** have **two different functions** for retrieving postings lists: one for when you don't care about positional information, and one for when you *do* want positional information in the returned postings. You will need to modify the `Index` interface with the second method, and clearly indicate which method loads positions and which does not. Your query engine must use the appropriate index retrieval method depending on the type of query: only a few Boolean query component types need positions (phrase queries, NEAR operator), whereas ranked queries only need $\text{tf}_{t,d}$ (and not the actual positions themselves).

Ranked retrievals: This is the biggest new requirement. Your main program must operate in two modes: Boolean query mode, and ranked query mode.

In ranked query mode, you must process a query without any Boolean operators and return the top $K = 10$ documents satisfying the query. Use the "term at a time" algorithm as discussed in class:

1. For each term t in the query:
   (a) Calculate $\text{w}_{q,t} = \ln\left(1 + \frac{N}{\text{df}_t}\right)$
   (b) For each document d in t's postings list:
      i. Acquire an accumulator value A_d (the design of this system is up to you).
      ii. Calculate $\text{w}_{d,t} = 1 + \ln(\text{tf}_{t,d})$.
      iii. Increase A_d by $\text{w}_{d,t} \times \text{w}_{q,t}$.
2. For each non-zero A_d, divide A_d by L_d, where L_d is read from the `docWeights.bin` file.
3. Select and return the top $K = 10$ documents by largest A_d value. (Use a binary heap priority queue to select the largest results; do **not** sort the accumulators.)

Use **8-byte floating point numbers** for all the calculations.

Printing ranked retrieval results: Please print the title of each document returned from a ranked retrieval, **as well as the final accumulator value for that document.**

Other features from Milestone 1: You must maintain all other features from Milestone 1, potentially having to update/re-engineer them to use the new indexing system. This includes the query language processor, the document processing rules, and any other optional features you completed.

Additional Requirements

Each of the following additional requirements are worth a certain amount of points. Your group must select and implement additional features worth **at least** P **total points** from the following list, where P is equal to...

- **two times** the number of people in your group...
- **plus** 2, if *at least one* member of your group is enrolled in CECS 529.

Options:

SPIMI algorithm - 5 points: Program the SPIMI algorithm for creating an on-disk index. Set some constant threshold value that determines when your in-memory index is "full". Process tokens into an in-memory positional inverted index until the index is full, then save the index to a "partial" index using your `DiskIndexWriter` class. Clear the index, then process tokens again, repeating the process until all documents are processed. Merge the partial indexes together into one final index using the SPIMI merge algorithm. You can simplify the merge algorithm in this way:

1. Assume that the entire vocabulary fits into main memory. Construct this full vocabulary by reading the vocabularies from each bucket index and unioning those together.
2. For each term in the sorted vocabulary, read the postings for that term from each partial index in order. Merge those postings into one final list, then write that list to disk.
3. Repeat.

In this way, you do not have to program the "priority queue" aspect of the SPIMI algorithm.

Modify your search engine to always use the SPIMI algorithm instead of the default `DiskIndexWriter` construction method. Demonstrate that your code works by indexing the **entire** `mlb-articles-full.zip` corpus on BeachBoard, which has 200,000 articles and takes the example search engine implementation 15 minutes to index with SPIMI.

Dynamic Indexing - 5 points: Implement a dynamic indexing strategy using logarithmic merging, so your index can handle changes to the corpus at run-time. You do not need to automatically detect these changes; you can use a menu/special query to select a directory that contains new documents to process. You must:

- code the logarithmic merging strategy: process tokens from documents and place them into an in-memory index. When the index grows to some pre-selected size, perform the logarithmic merge into indexes I_0, I_1,etc. as appropriate. (The merge algorithm will be very similar to SPIMI's, except there's only two indexes being merged at a time – instead of a great many in SPIMI.)
- modify your index to pull postings for a term from the in-memory index *and also* all the logarithmic indexes, and merge those results together before returning.

Spelling correction - 4 points: You can only attempt this if you also choose "K-gram index on disk" for 1 point.

Implement a spelling correction module for your search engine. Any time a user runs a query using a term that is either missing from the vocabulary or whose document frequency is below some threshold value (your decision), run the query and give results as normal, but then print a suggested modified query where the possibly misspelled term(s) is replaced by a most-likely correction. Ask the user if they would like to run this modified query. To select the most-likely correction:

1. Select all vocabulary types that have k-grams in common with the misspelled term, as described in lecture.
2. Calculate the Jaccard coefficient for each type in the selection.
3. For each type whose coefficient exceeds some threshold (your decision), calculate the edit distance from that type to the misspelled term.
4. Select the type with the lowest edit distance. If multiple types tie, select the type with the highest df_t (when stemmed).

Variant tf-idf formulas - 3 points: Researchers have spent considerable time testing other ways of calculating $\mathrm{w}_{q,t}$, $\mathrm{w}_{d,t}$, and L_d. Some formulations avoid the use of logarithms for efficiency; others use lessons learned from statistics and language processing to give more accurate results. For this option, you will configure your ranked retrieval engine so the user can select from multiple options for calculating $\mathrm{w}_{q,t}$, $\mathrm{w}_{d,t}$, and L_d.

These options can either be set at run time through a special menu, or by editing a configuration file that you read at program startup. (Your choice.)

You must support the four following weighting schemes:

Default	tf-idf	Okapi BM25	Wacky
$\mathrm{w}_{q,t} = \ln\left(1 + \frac{N}{\mathrm{df}_t}\right)$	$\mathrm{w}_{q,t} = \mathrm{idf}_t = \ln\frac{N}{\mathrm{df}_t}$	$\mathrm{w}_{q,t} = \max\left[0.1,\ \ln\left(\frac{N-\mathrm{df}_t+0.5}{\mathrm{df}_t+0.5}\right)\right]$	$\mathrm{w}_{q,t} = \max\left[0,\ \ln\frac{N-\mathrm{df}_t}{\mathrm{df}_t}\right]$
$\mathrm{w}_{d,t} = 1 + \ln\left(\mathrm{tf}_{t,d}\right)$	$\mathrm{w}_{d,t} = \mathrm{tf}_{t,d}$	$\mathrm{w}_{d,t} = \frac{2.2 \cdot \mathrm{tf}_{t,d}}{1.2 \cdot \left(0.25 + 0.75 \cdot \frac{\mathtt{docLength}_d}{\mathtt{docLength}_A}\right) + \mathrm{tf}_{t,d}}$	$\mathrm{w}_{d,t} = \frac{1+\ln(\mathrm{tf}_{t,d})}{1+\ln(\mathrm{ave}(\mathrm{tf}_{t,d}))}$
$L_d = \mathtt{docWeights}_d$	$L_d = \mathtt{docWeights}_d$	$L_d = 1$	$L_d = \sqrt{\mathtt{byteSize}_d}$

In the table:

- $\mathtt{docWeights}_d$ is the Euclidian weight of document d as described on page 1.
- $\mathtt{docLength}_d$ is the number of tokens in document d; $\mathtt{docLength}_A$ is the *average* number of tokens in all documents in the corpus.
- $\mathtt{byteSize}_d$ is the number of bytes in the file for document d.
- $\mathrm{ave}\left(\mathrm{tf}_{t,d}\right)$ is the average $\mathrm{tf}_{t,d}$ count for a particular document.

$\mathtt{docWeights}_d$, $\mathtt{docLength}_d$, $\mathtt{byteSize}_d$, and $\mathrm{ave}\left(\mathrm{tf}_{t,d}\right)$ are all per-document values, and each should be saved to the `docWeights.bin` file created during indexing. I recommend writing all four values in sequence for each document. You will also need to write $\mathtt{docLength}_A$ somewhere on disk, but it is a single value for the entire corpus.

Make sure you architect this system well. If your solution is a giant mass of "if-else" statements, well, *try something else*. Suggestion: look up the "strategy" design pattern from object oriented design / software engineering

Variable byte encoding - 3 points: Encode the index files using variable byte encoding.

Modify `IndexWriter` so all document ID gaps, position counts, and position gaps are written using variable byte encoding as described in lecture and in the book. Modify `DiskPositionalIndex` so it accounts for variable byte counts when reading postings and positions. Hint: you cannot assume that 4 bytes should be read for each int, so you now must read 1 byte at a time and decide whether you need more bytes for the number you are decoding. This also means you cannot seek/skip past positional postings if you don't need them (for non-phrase queries); you need to *scan* past them, counting the number of times you read a byte with a top-most bit of 1 and continuing to scan until you encounter one such byte for every position you are attempting to skip.

Your encoding and decoding methods must be efficient. You cannot use string variables in these operations; you must only work with integer types, bytes, and arrays of bytes.

DSP index - 2 points: Computing scores is relatively time consuming, especially when they involve logarithms. We can reduce the amount of computation needed to calculate ranking scores by precomputing $\mathrm{w}_{d,t}$ values and storing them in the index files, similar to how L_d values are precomputed and stored. We only compute $\mathrm{w}_{q,t}$ once per term in a query, so there's not much to gain by reading $\mathrm{w}_{q,t}$ values from disk... but we do thousands or millions of $\mathrm{w}_{d,t}$ scores for each term in a query, so precomputing these values can save a lot of work at runtime.

Add one additional 8-byte `double` value to each entry in the postings list in your disk-based positional index, after the document ID but before the $\mathrm{tf}_{t,d}$. That value should be the pre-computed $\mathrm{w}_{d,t}$ value for the document-term pair being written, for the definition of $\mathrm{w}_{d,t}$ in the mandatory requirements above. The postings for a term will then look like $d\ \mathrm{w}_{d,t}\ \mathrm{tf}_{t,d}\ p_1\ p_2\ \cdots\ p_i$ (for each document containing the term). Such an index is called a **DSP** (**D**ocuments, **S**cores, **P**ositions) in academic search literature.

Amend your postings retrieval routine to include the $\mathrm{w}_{d,t}$ value in the returned data. When using $\mathrm{w}_{d,t}$ in the scoring algorithm, do not compute $\mathrm{w}_{d,t}$ with the formula above; simply use the value returned from the disk index as part of the retrieved postings.

Do not attempt to variable-byte encode these values, as they are not integers.

If you also complete the Variant tf-idf option, you should pre-compute **all four** $\mathrm{w}_{d,t}$ options for each document-term pair, and write each (in any specific order) to the disk index prior to $\mathrm{tf}_{t,d}$. Each value should be returned when retrieving postings.

Unit testing framework - 2 points: If you did not implement this option in Milestone 1, you may choose to implement it now using the same instructions. You must also include some tests for ranked retrieval by hand-computing scores for a few selected documents on a few selected queries and testing your algorithms on those results.

K-gram index on disk - 1 point: **If** you chose wildcard queries for Milestone 1, you may attempt this option. If you **did not** select wildcard queries for Milestone 1, you may implement this requirement **in addition** to Milestone 1's requirements. If you do, you may count this as **4 points** total.

Save your wildcard index to disk, and incorporate wildcards into ranked retrieval queries.

Extend the "create an index" procedure of your program to **also** generate a disk-based wildcard index, and likewise extend the "query an index" procedure to load this wildcard index for use with wildcard queries. I will leave the design of this index up to you, but will gladly give input if you need it. You may create a design in which the entire wildcard index is read into and retained in memory for the duration of the search engine; you do not need to architect a system that reads wildcard information from the binary file each time a wildcard query is needed.

To incorporate wildcards into ranked retrievals, simply include every vocabulary type that matches the wildcard token in the ranking procedure. (Yes, this gives higher scores to documents that contain multiple different words matching the wildcard query. If you can come up with a better procedure, I welcome your proposal.)

Biword index on disk - 1 point: **If** you chose biword index for Milestone 1, you may attempt this option. If you **did not** select biword index for Milestone 1, you may implement this requirement **in addition** to Milestone 1's requirements. If you do, you may count this as **2 points** total.

Save your biword index to disk. Apply the logic in the `DiskIndexWriter` class(es) to saving your biword index, and similarly apply the logic of the `DiskPositionalIndex` class to reading your biword index. You will still only use the index to retrieve postings for 2-length phrase queries in **Boolean retrieval mode**.

Soundex index on disk - 1 point: **If** you chose Soundex index for Milestone 1, you may attempt this option. If you **did not** select Soundex index for Milestone 1, you may implement this requirement **in addition** to Milestone 1's requirements. If you do, you may count this as **3 points** total.

Save your Soundex index to disk. Apply the logic in the `DiskIndexWriter` class(es) to saving your Soundex index, and similarly apply the logic of the `DiskPositionalIndex` class to reading your Soundex index. You will still only use the index to satisfy `:author` queries in **Boolean retrieval mode**.

NOT queries - 1 point: If you did not implement this option in Milestone 1, you may choose to implement it now using the same instructions.

NEAR/K operator - 1 point: If you did not implement this option in Milestone 1, you may choose to implement it now using the same instructions.

Summary

Milestone summary. You must:

1. Operate in two modes: Build Index and Query Index modes.
2. Support two querying styles: Boolean and Ranked.
3. Maintain all processing rules from Milestone 1.
4. Perform ranked retrievals using specified formulas.

You must choose additional features to implement according to your group size.