Johan Boye, KTH, 2022-01-13

DD2476 Search Engines and Information Retrieval Systems

Assignment 1: Boolean Retrieval[1]

The purpose of Assignment 1 is to learn how to build an inverted index. You will learn 1) how build a basic inverted index; 2) how to handle multiword queries; 3) how to handle phrase queries; 4) how to evaluate a search system; and 5) techniques for handling large indexes.

The recommended reading for Assignment 1 is that of Lectures 1-3.

Assignment 1 is graded, with the requirements for different grades listed below. In the beginning of the oral review session, the teacher will ask you what grade you aim for, and ask questions related to that grade. The assignment can only be presented once (unless you get an F) – you cannot raise your grade by doing additional tasks after the assignment has been examined and given a grade. ***Come prepared to the review session!*** *The review will take 15 minutes or less, so have all papers in order.*

E: *Completed Task 1.1-1.4 with some mistakes that could be corrected at the review session, completed Task 1.5 and 1.6.*
D: *E + completed Task 1.1-1.4 without mistakes.*
C: *E + completed the "C" variant of task 1.7.*
B: *E + completed the "B" variant or task 1.7.*
A: *B + completed task 1.8.*

These grades are valid for the review on February 8, 2022. See the Canvas pages for grading of delayed assignments.

Assignment 1 is intended to take around 40h to complete.

Dataset

In the three assignments in this course, you will implement a rudimentary search engine. The engine will be evaluated on a corpus of linked documents, the wiki for the US town Davis. The wiki can be found on **`https://daviswiki.org`**, and we also provide a cleaned" version without images or formatting, which is found in the course directory, see below.

NOTE: The wiki is constantly changing. There are therefore discrepancies between the cleaned copy (which is from 2014) and the online wiki.

[1] With contributions by Carl Eriksson, Dmytro Kalpakchi, Jussi Karlgren, and Hedvig Kjellström.

Computing Framework

The code and datasets needed for the assignment can be downloaded from Canvas. The datasets can also be accessed from the course directory `/info/DD2476/ir22/lab` on the school's UNIX system.

The **`ir`** directory contains the source code skeleton which you will use as the basis for the assignments 1-3. The code of each assignment will build on that of the previous assignment.

If you are using your own computer, you will need to download the **`davisWiki`** and **`guardian`** datasets from Canvas as well.

When everything is in place, compile the lab skeleton as follows:

```
sh compile_all.sh
```

If you are working on a Windows platform, you can instead use the **`bat`** files:

```
compile_all.bat
```

You might see some warnings; these may be ignored.

Task 1.1: Tokenization

The **`Tokenizer`** java class (found in the **`ir`** directory) performs tokenization of text. In its default setting, the program considers whitespace and punctuation as delimiters. This means, for example, that **`24/7`** will be considered as the tokens **`24`** and **`7`**. However, one can configure the tokenizer to deal with non-standard tokens (such as dates, phone numbers, mail addresses, etc.) by adding regular expressions in the file **`patterns.txt`**: If the next string of **non-whitespace** characters to be read from input matches one of the regular expressions in this file, the string is considered to be a single token. For instance, adding the pattern **`\d+/\d+`** to **`patterns.txt`** causes the program to emit **`24/7`** as a single token. **Your task is now to add regular expressions to `patterns.txt` so that the program handles non-standard tokens correctly.** Try to make your regular expressions general, so that each expression covers a class of non-standard tokens (e.g. dates). Try out your expressions on **`token_test.txt`** using the script **`run_tokenizer.sh`** (or **`run_tokenizer.bat`** on Windows). Your task is done when the output of the program is identical to the contents of the file **`token_test_tokenized_ok.txt`** (use the Unix **`diff`** utility to verify this). Less than 20 regular expressions should be required.

NOTE: files produced on Windows terminate lines using a carriage return character **`\r`** (ASCII code 13) in addition to a newline character **`\n`** (ASCII code 10), hence you might want to add **`--strip-trailing-cr`** option to **`diff`** to ignore carriage return chars.

At the review

To pass Task 1.1, you should be able to run the tokenizer using your regular expressions, and let the teacher verify that the output is correct. If asked, you should also be able to explain the purpose of a particular regular expression.

Task 1.2: Basic Inverted Index

When we have sorted out the tokenization issues, we can start working on index construction. To start the search engine, write

```
sh run_search_engine.sh
```

or, on a Windows platform:

```
run_search_engine.bat
```

Note that if you are using your own computer, you might need to edit the script and change the path to the **`davisWiki`** data set.

You should now see the GUI where search queries are entered, and search results presented. The **`-d`** flag on the line above tells the program which directory to index. You might specify any number of directories by

```
-d directory1 -d directory2 -d directory3
```

etc. The program indexes all text files in the specified directories.

Enter some words in the search text box and press Enter. You will now see in the result text area:

```
Found 0 matching document(s)
```

In order for the system to actually find some matching documents, you will now need to add code to some of the classes in the **`ir`** directory. The **`processFiles`** method in the **`Indexer`** class reads files and produces a stream of tokens to be indexed. However, currently the program is incomplete, and no index is built. **Your task is to complete the classes `HashedIndex`, `PostingsList`, `PostingsEntry` and `Searcher` (and possibly add classes of your own), so that the program builds an inverted index, and can search the index for 1-word queries.**

When you have finished adding to the program, compile and run it, indexing the **`davisWiki`** data set. Try the search queries

zombie

which should result in **36** retrieved documents, and

attack

which should result in **228** retrieved documents.

At the review

There will not be any examination of Task 1.2, it is merely a preparation for Task 1.3.

Task 1.3: Multiword Queries

The search engine implemented in Task 1.2 can only handle single word queries; it will now be extended to queries consisting of several words.

Extend the **`search`** method in the **`Searcher`** class **to implement the intersection algorithm** (page 11 in the textbook), so that you can do multiword searches.

When you have finished adding to the program, compile and run it, indexing the **`davisWiki`** data set. Try the search queries

zombie attack

which should result in **15** retrieved documents,

money transfer

which should result in **105** retrieved documents,

a cell phone

which should result in **195** retrieved documents,

what they are selling

which should result in **214** retrieved documents.

Furthermore, design your own query (or queries) with 4 words or more, and try it on the dataset.

At the review

To pass Task 1.3, your search engine must return correct results on the queries above, and on any query specified by the teacher. Indexing the **`davisWiki`** corpus must not take more than 3 minutes. Search **must be immediate** (definitely less than 0.1s) for any search query. You should also be able to explain all parts of the code that you edited, draw the data structure on paper, and explain from that figure how an intersection query is executed.

Task 1.4: Phrase Queries

Modify your program so that it is possible to search for contiguous phrases like "money transfer" using the techniques described in Section 2.4.2 in the textbook. Only documents that contain that exact phrase should be returned; documents that include the words at separate places should **not** be returned. Note that the algorithm on page 39 is **not** exactly what you are looking for.

You will need to add code to the **`search`** method in the **`Searcher`** class, so that when this method is called with the **`queryType`** parameter having the value **`QueryType.PHRASE_QUERY`**, the search query should be treated as a phrase query. When **`search`** is called with the value **`QueryType.INTERSECTION_QUERY`**, the program should behave as in Task 2 above.

Compile the program and run it, indexing the **`davisWiki`** data set. At runtime, you can choose between intersection queries and phrase queries in the "Search options" menu. Choose the "Phrase query" option and try the same search queries as before,

zombie attack

which should result in **14** retrieved documents,

money transfer

which should result in **2** retrieved document,

a cell phone

which should result in **30** retrieved documents,

what they are selling

which should result in **2** retrieved documents.

Run your own query/queries as phrase query.

At the review

To pass Task 1.4, your search engine must return correct results on the queries above, and on any query specified by the teacher. Indexing the **`davisWiki`** corpus must not take more than 3 minutes. Search **must be immediate** (definitely less than 0.1s) for any search query. You should also be able to explain all parts of the code that you edited, draw the data structure on paper, and explain from that figure how a phrase query is executed.

You should also prepare an answer to the question *Why are fewer documents generally returned in phrase query mode than in intersection query mode?*

Task 1.5: What is a good search result?

The objective of this task is to **reflect on what constitutes a good answer to a query**. Run the program from Task 1.4, indexing the data set **`davisWiki`**. Choose the "Intersection query" option in the "Search options" menu.

Search the indexed data sets with the following query:

graduate program mathematics

Go through the list of 22 returned documents and rate their relevance for the query. (By clicking on the file name in the search results window, you can see the contents of the file in the lower window of the GUI.) Use the following rating scale:

(0) Irrelevant document. The document does not contain any information about the topic.
(1) Marginally relevant document. The document only points to the topic. It does not contain more or other information than the topic description.
(2) Fairly relevant document. The document contains more information than the topic description, but the presentation is not exhaustive.
(3) Highly relevant document. The document discusses the themes of the topic exhaustively.

[E. Sormunen. Liberal relevance criteria of TREC—Counting on negligible documents? *ACM SIGIR*, 2002]

Edit the results into a **plain text file** (with a '.txt' extension) using the following space-separated format, one line per assessed document:

`QUERY_ID DOC_ID RELEVANCE_SCORE`

where `QUERY_ID` = 1, `DOC_ID` = the name of the document, `RELEVANCE_SCORE` = [0, 1, 2, 3]. Upload the file to Canvas under 'Assignment 1'**.**

It should be noted that there is no objectively correct relevance label for a certain query-document combination! It is a matter of judgment. For difficult cases, write a short note (not in the labeling file, but a separate note) on why you chose the label you did. At the review, you will present three difficult cases.

Compute the **precision** (relevant documents = documents with relevance > 0) and **recall** (up to an unknown scale factor) of the returned list. To get a recall estimate up to an unknown scale factor, assume the total number of relevant documents in the corpus to be 100.

At the review

To pass Task 1.5, you should show the text file with labeled documents, in the correct format. Before the review, upload the file to Canvas under 'Assignment 1'. You should be able to explain the concepts precision and recall and give account for the precision and recall (up to a scale factor) of the returned document list.

You should also, for at least 3 documents in the list, be able to expand on why you chose the labels you did.

NOTE: You will not be failed in the review for selecting the "wrong" labels. However, you might be asked to redo this task if you are unable to motivate your labels in a satisfactory manner.

Task 1.6: What is a good query?

In this task, you will again use the same search settings as in Task 1.5. The objective is here to **reflect on how a user formulates a query based on an information need**, and how the query is merely an incomplete representation of the latent information need.

Consider the following information need:

Info about the education in Mathematics on a graduate level at UC Davis

Design a query which you think will return documents relevant to the information need. Do an intersection search using this query, and

1) look at the number of documents – if there are more than 25, the query is too general, if there are fewer than 2, the query is too specific.
2) when there is a reasonable number (<25), label the returned documents as "relevant" or "non-relevant" in exactly the same manner as in Task 1.4. **Remember to save query and returned list for the review**.

Now, go back to the information need description and ponder if you can improve the precision and recall of the returned list by removing, adding or exchanging words in the query. Do a search using the new query and see if a higher share of relevant documents were returned. Repeat this until you have found an "optimal" query for the information need description.

At the review

To pass Task 1.6, you should show the optimized query, as well as the full list of query variants leading up to the final query. You should also expand on why you think that the final query gave better precision and/or recall than the earlier variants.

Furthermore, prepare a comprehensive answer to the question *Why can we not simply set the query to be the entire information need description?*

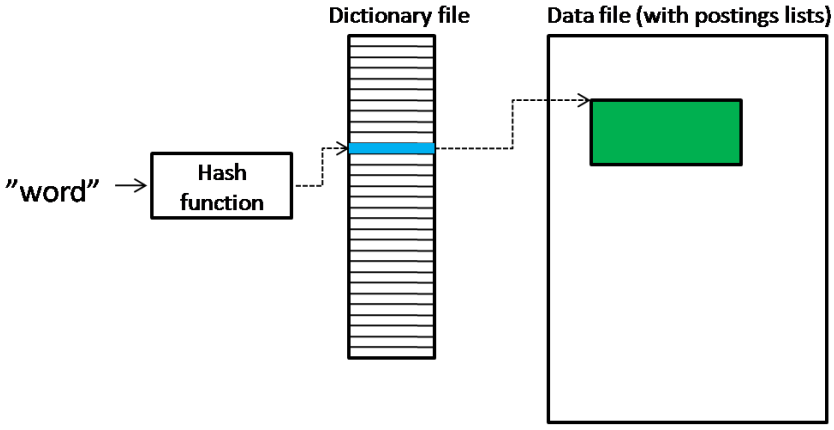
Task 1.7: Inverted index as a hash table on disk (C-B)

In realistic applications, we of course cannot index the whole document collection every time we start the search engine. Moreover, the complete index would be too large to fit in working memory. In this task and the next, you will implement a more scalable solution.

For grade C (not encouraged!) we will accept any solution that indexes the davisWiki corpus and saves the index to disk, such that:

- One should be able to stop the search engine and restart it again, and the engine should be able to process search queries immediately, without any startup time.
- The results of a search query should be instantly produced.
- Of course, all search results should be the same those of task 1.3 and 1.4.
- No object serialization or external libraries are used.

For grade B to A (encouraged!), you will implement the index by means of a persistent hash table on disk. A difficulty is that the postings list for different words can be of very different sizes; very large for common words like "the", and very small for unusual words. Because of this, we will adopt the following organization. The persistent hash table will reside in two files: the data file (containing all the postings lists), and the dictionary file (containing a representation of the terms/words, and pointers into the data file).



The figure above shows the intended organization. Given a word *w* (e.g. "word"), we compute a hash value *h*(*w*) for that word. The hash value points out the address in the dictionary file, where we put an entry containing a pointer into the data file to the address where the postings list is stored + possibly some more information you deem necessary.

Crucially, the entries in the dictionary file **must all be of the same size**! The entries in the data file, on the other hand, can be of any size.

For grade B, first change the declaration of the **`index`** variable in **`Engine.java`** to look like this:

```
Index index = new PersistentHashedIndex();
```

and recompile. Your task is then to add code to the **`PersistentHashedIndex`** class, so that it works as follows:

- All terms are first indexed, using the **`insert`** method, the same way as in the ordinary **`HashedIndex`** class, and put in the internal hash map **`index`** (just as in task 1.4).
- When all tokens are processed, the method **`writeIndex`** is called. It should go through all the terms in **`index`**, and for every term *w* write a string representation of its postings list to the first available memory location in the data file (don't serialize the postings list objects; it will make the file 20 times bigger, compared to a simple string representation). This memory location should be stored in an entry, which is stored in the dictionary file at the location given by *h*(*w*) using the method **`writeEntry`**.
- For this to work, you need to devise a hash function that, given a term *w*, returns an integer between 0 and the size of the dictionary file (given by the constant **`TABLESIZE`**). A good value for a dictionary file size is around 3-4 times bigger than the number of unique terms to be hashed, which will result in a number around 600000 for **`davisWiki`**. (Should the table size be a specific kind of number, or is any number fine? Why?)
- Note that there will be hashing collisions, and you will have to take care of these.
- To make search work, you need to complete the method **`getPostings`**, so that given a word w, the correct entry in the dictionary file is located, which will then point to the correct address in the data file where the postings list is stored.

- When everything works as it should, you should be able to index the **`davisWiki`** corpus using the start script **`run_search_engine.sh`** (or the script **`run_search_engine.bat`**), stop the search engine, and restart it using **`run_persistent.sh`** (or **`run_persistent.bat`**). The search engine should now start immediately without indexing and the search time should still be close to instant.

At the review

To pass Task 1.7, you should be able to index **`davisWiki`** corpus, save it on disk, then restart the program without indexing and still get correct results on the queries in Tasks 1.3 and 1.4. Indexing the **`davisWiki`** corpus must not take more than 3 minutes. Search **must be immediate** (definitely less than 0.1s) for any search query. You should also be able to explain how your implementation of a persistent hash table works and justify your choice of a hash function for words.

Task 1.8 Merging disk indexes (A)

A big assumption made in task 1.7 is that all tokens with the respective postings lists can be fit into the main memory, which is clearly not the case in real-life applications. In this task, you will work with the **`guardian`** corpus which cannot be indexed with the method adopted in the previous task. Instead, in task 1.8 you will generate the index as multiple (partial) dictionary and data files, and then merge them together into one large persistent index, i.e., one dictionary file and one data file. For this task, it might be handy to create a new class called **`PersistentScalableHashedIndex`** (you might want to subclass the **`PersistentHashedIndex`** class to reuse as much code as possible).

One way of solving the problem is to modify the **`insert`** and **`cleanup`** methods of **`PersistentHashedIndex`** class, so that the program works as follows:

- The terms are indexed as in Task 1.7 up to a specific point, e.g., a specific number of tokens processed.
- When this specific point is reached, the index is stored on disk and the main-memory hash map **`index`** along with hash tables **`docNames`** and **`docLengths`** are cleared. The pointer **`free`** to next unoccupied bucket in the data file is reset to 0. The random-access files **`dictionaryFile`** and **`dataFile`** are recreated.
- If at least two intermediate indexes have been written, i.e., two dictionary files and two data files, a thread for merging them is started as a background thread. The merge algorithm depends totally on the representation of the records in both dictionary and data files as well as a naming convention you choose to adopt for intermediate indexes. Think if you need to create/split any additional files to make the merging process easier and/or more reliable (one particular file you don't want to forget about is the **`docInfo`** file).
- The indexing of the next portion of documents continues in the same fashion as stated in the step 1.

- After the last intermediate index was written on disk the program should wait for all the threads to be finished in the **`cleanup`** method. When all the threads are finished, the random-access files **`dictionaryFile`** and **`dataFile`** should point to the final merged dictionary and data file respectively.

Hints:

- When indexing the Guardian dataset, it is recommended to increase the constant **`TABLESIZE`** to a number close to 3,500,000.
- There exists a clear tradeoff between the speed of indexing and merging and the size of the intermediate index files. It might be a good idea to split an index into about 5-6 intermediate files.

Here are some search examples from the Guardian corpus:

Query	Intersection search results	Phrase search results
zombie attack	15	1
a cell phone	104	3
money transfer	573	25
what they are selling	1524	1

At the review

To pass Task 1.8, you should be able to index **`davisWiki`** corpus, save it on disk while indexing and demonstrate how your merging algorithm works on the go, then restart the program without indexing and still get correct results for the queries in Tasks 1.3 and 1.4. You should also be able to search the **`guardian`** corpus using the index, saved previously on your disk, get the correct results on the queries above and on any query specified by the teacher. The indexing of the whole **`guardian`** corpus will take some time (probably not more than 30 minutes); however, you should be able to demonstrate the indexing up to the first merge of the two intermediate indexes at the review.